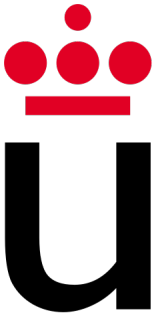

Universidad
Rey Juan Carlos

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA DEL SOFTWARE

Curso Académico 2021/2022

Trabajo Fin de Grado

DISEÑO Y DESARROLLO DE UNA APLICACIÓN PARA LA SIMULACIÓN DE ECOSISTEMAS

Autor: Daniel Carmona Pedrajas

Director: Carlos Enrique Cuesta Quintero

Índice general

1 Introducción **1**

1.1 Idea general 1

1.2 Motivación 2

2 Objetivos **3**

2.1 Objetivo general 3

2.2 Objetivos específicos 3

3 Estado del arte **5**

3.1 Sistemas predador-presa 5

3.2 El modelo Lotka-Volterra 6

3.3 Simulación con autómatas celulares 8

3.3.1 Automátas celulares 8

3.3.2 Ejemplo de simulación basada en autómatas celulares 10

3.4 Motores de simulación 12

4 Contexto de desarrollo **13**

4.1 Metodología 13

4.2 Motor gráfico Unity 15

4.3 Lenguaje de programación C# 16

5 Funcionamiento de la simulación **19**

5.1 Flujo de la simulación 19

5.2 Fase de supervivencia 20

5.3 Fase de evolución 20

5.3.1 Ciclo de vida 20
5.3.2 Reproducción 21

6 Principios SOLID **23**

6.1 Principio de Responsabilidad Única 23
6.2 Principio Abierto-Cerrado 24
6.3 Principio de Sustitución de Liskov 24
6.4 Principio de Segregación de Interfaces 24
6.5 Principio de Inversión de Dependencias 24

7 Patrones de diseño **25**

7.1 ¿Qué son los patrones de diseño? 25
7.2 Patrones de creación 26
7.2.1 Patrón Builder 26
7.2.2 Patrón Factory Method 27
7.3 Patrones estructurales 28
7.3.1 Patrón Modelo-Vista-Controlador 28
7.3.2 Patrón Composite 29
7.4 Patrones de comportamiento 30
7.4.1 Patrón State 30
7.4.2 Patrón Strategy 31
7.4.3 Patrón Mediator 32
7.5 Patrones de optimización 33
7.5.1 Patrón Spatial Partition 33

8 Etapa 0: Planteamiento inicial **35**

8.1 Objetivo 35
8.2 Diseño 35
8.2.1 Modelo 36
8.2.2 Vista 38
8.2.3 Controlador 39

9 Etapa 1: Prototipo **41**

9.1 Objetivos . . . 41

9.2 Estado de la simulación . . . 42

9.3 Diseño . . . 43

9.3.1 Modelo . . . 43

9.3.2 Vista . . . 48

9.3.3 Controlador . . . 49

9.4 Principios SOLID . . . 49

9.4.1 Principio de Responsabilidad Única . . . 49

9.4.2 Principio abierto/cerrado . . . 50

9.4.3 Principio de Sustitución de Liskov . . . 50

9.4.4 Principio de Segregación de Interfaces . . . 51

9.4.5 Principio de Inversión de Dependecias . . . 51

9.5 Resultados . . . 51

9.5.1 Experimento 1: Extinción . . . 51

9.5.2 Experimento 2: Equilibrio . . . 52

9.6 Refactorizaciones . . . 53

10 Etapa 2: Implementación de estrategias de caza **55**

10.1 Objetivos . . . 55

10.2 Estado de la simulación . . . 56

10.3 Diseño . . . 57

10.3.1 Modelo . . . 57

10.3.2 Vista . . . 64

10.3.3 Controlador . . . 64

10.4 Principios SOLID . . . 65

10.4.1 Principio de Responsabilidad Única . . . 65

10.4.2 Principio de abierto/cerrado . . . 65

10.4.3 Principio de Sustitución de Liskov . . . 65

10.4.4 Principio de Segregación de Interfaces . . . 65

10.4.5 Principio de Inversión de Dependencias . . . 65

10.5 Resultados 66
10.5.1 Caza en grupo 66
10.5.2 Caza individual 68
10.6 Refactorizaciones 69

11 Conclusiones y trabajos futuros **71**
11.1 Conclusiones 71
11.2 Trabajos futuros 72

A Diagramas UML completos **i**
A.1 Etapa 0 i
A.2 Etapa 1 iii
A.3 Etapa 2 v

B Manual de usuario **vii**
B.1 Descarga e instalación vii
B.2 Menú de configuración viii

Bibliografía **xi**

Índice de figuras

3.1 Gráficas del modelo Lotka-Volterra 7
3.2 Gráfico de cambios relativos en la densidad predador-presa 8
3.3 Ejemplos de los cuatro comportamientos básicos de los AC 9
3.4 Niveles de población de todas las especies, normalizadas para su comparación 10

4.1 Ciclo iterativo incremental 14
4.2 Porcentaje de usuarios por lenguaje de programación 17

5.1 Diagrama de flujo de la simulación 19

7.1 Diagrama UML Builder 26
7.2 Diagrama UML Factory Method 27
7.3 Estructura del patrón Modelo-Vista-Controlador 28
7.4 Diagrama UML Composite 29
7.5 Diagrama UML State 30
7.6 Diagrama UML Strategy 31
7.7 Diagrama UML Mediator 32
7.8 Diagrama UML Spatial Partition 33

8.1 Diagrama UML del modelo 37
8.2 Diagrama UML de la vista 38
8.3 Diagrama UML del controlador 39

9.1 Diagrama UML de Ecosystem en la etapa 1 44
9.2 Diagrama UML de AnimalGroup en la etapa 1 45
9.3 Diagrama UML de Animal y AnimalState en la etapa 1 46

9.4 Diagrama UML de Vec3 en la etapa 1 47
9.5 Diagrama UML de View en la etapa 1 48
9.6 Diagrama UML de AnimalGroupView y AnimalView en la etapa 1 48
9.7 Diagrama UML de Controller en etapa 1 49
9.8 Gráfico de poblaciones del experimento 1 52
9.9 Gráfico de poblaciones del experimento 2 53

10.1 Diagrama UML de Ecosystem en la etapa 2 58
10.2 Diagrama UML de AnimalGroup en la etapa 2 58
10.3 Diagrama UML de Animal en la etapa 2 60
10.4 Diagrama UML de AnimalFleeState en la etapa 2 61
10.5 Diagrama UML de AnimalHuntState en la etapa 2 63
10.6 Diagrama UML de Vec3 en la etapa 2 64
10.7 Diagrama UML de Controller en la etapa 2 64
10.8 Gráfico de poblaciones del experimento 1 67
10.9 Gráfico de poblaciones del experimento 2 68
10.10Gráfico de poblaciones del experimento 3 69

11.1 Posible diseño para la funcionalidad de reproducción 73

A.1 Diagrama UML del sistema en la etapa 0 ii
A.2 Diagrama UML del sistema en la etapa 1 iv
A.3 Diagrama UML simplificado del sistema en la etapa 2 vi

B.1 Guía de descarga vii
B.2 Abrir el archivo viii
B.3 Menú de configuración viii

Índice de tablas

3.1 Pseudocódigo ejemplo de autómatas celulares 11

9.1 Parámetros Experimento 1, Etapa 1 51
9.2 Parámetros Experimento 2, Etapa 1 52

10.1 Parámetros Animales Experimento 1, Etapa 2 66
10.2 Parámetros Plantas Experimento 1, Etapa 2. 66
10.3 Parámetros Animales Experimento 2, Etapa 2 67
10.4 Parámetros Plantas Experimento 2, Etapa 2. 67
10.5 Parámetros Animales Experimento 3, Etapa 2 68
10.6 Parámetros Plantas Experimento 3, Etapa 2. 68

Capítulo 1

Introducción

En este capítulo se explica cuál es la idea general del proyecto y su origen.

1.1 Idea general

El principal objetivo de este proyecto es crear una aplicación que simule la dinámica poblacional de dos especies que conviven en un ecosistema. Una de las especies será la especie presa y la otra será la especie depredadora. Para simular la relación entre ellas se modelarán distintas estrategias de caza.
La aplicación se desarrollará por etapas, explicando las decisiones de diseño tomadas y visualizando los datos obtenidos de las simulaciones y experimentos en cada etapa para detectar propiedades emergentes y así definir los objetivos y funcionalidades a implementar en la siguiente fase de desarrollo.
Es importante remarcar que este documento será complementado por la memoria del TFG de Matemáticas con título "*SIMULACIÓN DE ESTRATEGIAS DE CAZA DE ANIMALES GREGARIOS A PARTIR DE METAHEURÍSTICAS*" ya que en ella se detallará el funcionamiento de los algoritmos que se han implementado para modelar las estrategias de caza.

1.2 Motivación

Siempre he estado interesado en los animales, recuerdo ir de pequeño a casa de mi abuela y pedirla que me pusiera documentales en vez de dibujos animados. Con el paso del tiempo empecé a percibir una cierta belleza en lo cruda que es la naturaleza, no hay ni buenos ni malos, simplemente seres vivos luchando por su supervivencia.

Cuando cursé Fundamentos Biológicos en el primer año de carrera realizamos una práctica que simulaba la competición por los recursos entre especies usando una hoja de cálculo. Fue en ese momento que reparé en la relación evidente que tienen las matemáticas con cualquier otro campo del conocimiento y en el potencial que hay en la fusión de las distintas disciplinas. Siempre se han hecho grandes avances cuando esto ha ocurrido, por ejemplo, *El origen de las especies* nació de la unión entre biología y geología y rompió con todas las teorías de evolución anteriores, lo que lo ha convertido en uno de los textos científicos más importantes de todos los tiempos.

Es por todo esto que he decidido usar todo lo aprendido en el Doble Grado en Ingeniería del Software y Matemáticas y lo poco que sé de biología para hacer este TFG. Tengo claro que no supondrá ninguna revolución aunque espero que salga algo interesante de aquí.

Capítulo 2

Objetivos

En este capítulo se expondrán los objetivos del proyecto.

2.1 Objetivo general

Crear una aplicación que permita la simulación de un ecosistema.

2.2 Objetivos específicos

1. Diseñar la arquitectura del sistema.
2. Implementar la arquitectura del sistema en iteraciones.
3. Analizar los resultados de cada iteración.
4. Identificar propiedades emergentes del sistema.

Capítulo 3

Estado del arte

Este capítulo se centra en desarrollar el estado del arte sobre sistemas predador-presa y no se pretende hacerlo sobre el software de simulación, ya que es un campo muy complejo sobre el que versan libros enteros como podría ser *Building Software for Simulation* [1] de James J. Nutaro.
Se comienza repasando los distintos métodos de simulación que existen para los sistemas predador-presa. En primer lugar se explica el concepto de sistema predador-presa, para luego mostrar dos modos de simularlos: basados en modelado matemático con ecuaciones diferenciales y basados en autómatas celulares.
Se puede completar este estado del arte consultando el capítulo 4 del TFG de Matemáticas que versa sobre modelado de estrategias de caza.

3.1 Sistemas predador-presa

Los sistemas predador-presa [2] simulan un tipo de interacción productor-consumidor en el que participan dos especies, una depredadora de la otra. Este tipo de sistemas son característicos por tener procesos autorreguladores que les empujan a encontrar un equilibrio. Este equilibrio es fácil de conseguir porque el estado del sistema no depende del estado de un único individuo, sino que cada parte del sistema depende de la otra para estabilizarse (si la población de presas disminuye, los predadores morirán de hambre lo que hará que las presas tengan menor probabilidad de ser cazadas y su población crecerá hasta llegar a un nivel anterior).

3.2 El modelo Lotka-Volterra

Es el modelo [3] más simple de un sistema predador-presa. Fue desarrollado de forma independiente por Lotka (1925) y Volterra (1926) cuando repararon en la naturaleza cíclica de la dinámica de poblaciones.

El modelo se basa en un sistema de dos ecuaciones diferenciales de primer orden:

$$\frac{dH}{dt} = H(r - aP) \tag{3.1}$$

$$\frac{dP}{dt} = P(bH - m) \tag{3.2}$$

Donde:

- t es el tiempo.
- $H(t)$ es la función de densidad de población de las presas en un instante t.
- $P(t)$ es la función de densidad de población de los depredadores en un instante t.
- r es el ratio de crecimiento natural de población en ausencia de predación.
- a es el ratio de predación.
- b es el ratio de reproducción de los predadores por presa cazada.
- m es el ratio de mortalidad de predadores.

Podemos resolver estas ecuaciones diferenciales aplicando el método numérico de Runge-Kutta de grado 2 dadas unas condiciones iniciales $H(t_0) = H_0$ y $P(t_0) = P_0$ y un intervalo de tiempo $I = (0, z)$ obteniendo así las siguientes aproximaciones:

$$H_{n+1} = H_n + \frac{\Delta t}{2}\Big(k_1 + k_2\Big) \qquad n \in \{0, 1, ..., N-2\} \tag{3.3}$$

$$P_{n+1} = P_n + \frac{\Delta t}{2}\Big(k_3 + k_4\Big) \qquad n \in \{0, 1, ..., N-2\} \tag{3.4}$$

Donde:

$$k_1 = H_n(r - aP_n) \tag{3.5}$$

$$k_2 = (H_n + \Delta t k_1)(r - aP_n) \tag{3.6}$$

$$k_3 = P_n(bH_n - m) \tag{3.7}$$

$$k_4 = (P_n + \Delta t k_3)(bH - m) \tag{3.8}$$

$$N = \frac{z}{\Delta t} \tag{3.9}$$

A continuación se muestran dos ejemplos del modelo que usan los mismos parámetros pero tienen distintas condiciones iniciales:

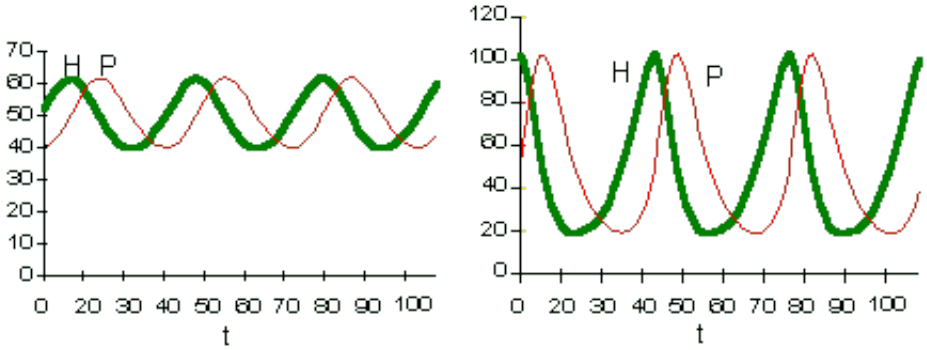

Figura 3.1: Gráficas del modelo Lotka-Volterra. Fuente: [1]

Se aprecia que el modelo no tiene estabilidad asintótica, no converge en un punto atractor (no "olvida" las condiciones iniciales).

En la siguiente figura cada punto $Q = (H(t), P(t))$ del plano tiene unas coordenadas que reflejan la densidad de población de las presas y los depredadores en un instante t determinado.

Observamos trayectorias cerradas que reflejan la naturaleza cíclica de la relación predador-presa.

El modelo Lotka-Volterra no es demasiado realista ya que no considera la competición por recursos entre predadores o presa, además asume que todos los individuos de una

[1] https://web.ma.utexas.edu/users/davis/375/popecol/lec10/lotka.html

[2] https://web.ma.utexas.edu/users/davis/375/popecol/lec10/lotka.html

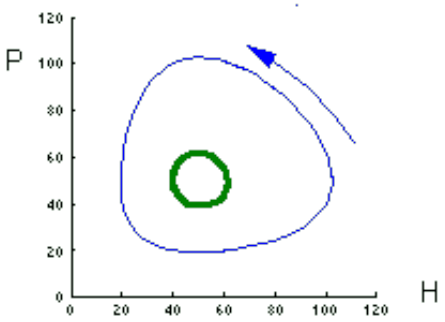

Figura 3.2: Gráfico de cambios relativos en la densidad predador-presa. Fuente: [2]

especie son idénticos. Como resultado de esto, la población de las presas podría crecer de forma infinita sin una limitación de recursos. Lo mismo ocurre con los predadores al tener un ratio de consumo ilimitado. El ratio del consumo de los predadores es proporcional a la densidad de población de las presas por lo que no es sorprendente que el comportamiento del modelo no sea natural y no muestre estabilidad asintótica.

3.3 Simulación con autómatas celulares

Una aproximación al modelado de relaciones predador-presa son los sistemas basados en individuos. Estos sistemas [4] están compuestos por unidades muy simples que interactúan entre ellas siguiendo unas reglas. El nivel de interacción entre estos agentes determina parcialmente el comportamiento general del sistema dando lugar a lo que se conoce como propiedades emergentes. De estas propiedades han nacido simulaciones como el famosísimo *Game of Life* de Conway que con simples reglas da lugar a patrones de células que pueden llegar a replicarse o avanzar por el espacio celular como si tuvieran vida propia.

3.3.1 Automátas celulares

Un autómata celular [5] (AC a partir de ahora) es un sistema computacional abstracto y discreto.

Primeramente los AC son **discretos** espacial y temporalmente, espacialmente porque están compuestos de un número finito de unidades homogéneas llamadas átomos o células

y temporalmente porque en cada unidad de tiempo cada célula puede estar en un estado perteneciente a un conjunto finito (el estado en el que se encuentra una célula afecta al estado de las células vecinas).
En segundo lugar, los AC son **abstractos**, es decir, pueden ser especificados en términos matemáticos.
Por último, los AC son sistemas **computacionales** porque son capaces de resolver problemas algorítmicos ya que pueden emular una máquina de Turing.

En los años 80 Stephen Wolfram resucitó sin apenas ayuda el estudio de los AC, abordande el tema con firme análisis matemático. Una de sus aportaciones fue la clasificación de los AC, que dividió en cuatro clases [6]:

- **Clase 1**: El comportamiento es muy simpe y casi todas las condiciones iniciales conducen al mismo estado final.
- **Clase 2**: Hay una gran cantidad de posibles estados finales, pero todos ellos consisten de un conjunto concreto de estructuras que permanecen igual para siempre o se repiten cada pocos pasos.
- **Clase 3**: El comportamiento es más complicado, parece aleatorio en muchos aspectos, aunque se aprecian pequeñas estructuras triangulares en ciertos niveles.
- **Clase 4**: Hay una mezcla de orden y aleatoriedad: las estructuras locales son simples, pero se mueven e interaccionan con las demás de forma compleja.

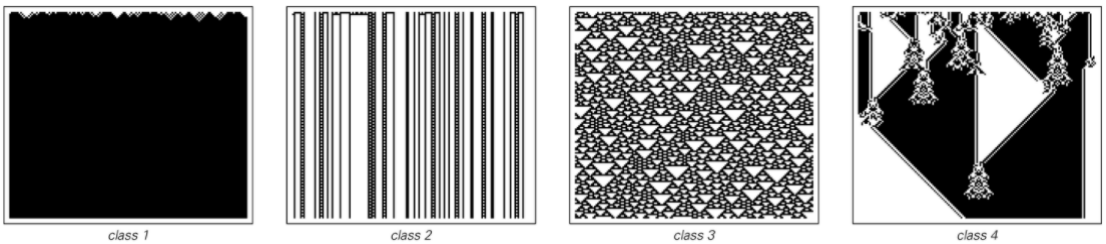

Figura 3.3: Ejemplos de los cuatro comportamientos básicos de los AC.

[2]Stephen Wolfram, "A New Kind of Science: Four Classes of Behavior: Image Source for Page 231" from the Notebook Archive (2018), `https://notebookarchive.org/2018-12-1g671ci`

El poder de los autómatas celulares es infinito, existen libros que profundizan sobre ellos como por ejemplo *Cellular Automata* [7] de E. F. Codd donde se habla sobre la computabilidad en espacios celulares y que presenta el diseño de una máquina capaz de ejecutar cualquier función computable[3].

3.3.2 Ejemplo de simulación basada en autómatas celulares

Este ejemplo [8] consta de una cuadrícula con una determinada altura h y longitud l, por lo que el AC constará de $c = h * l$ células. En cada unidad de tiempo t una célula tiene un estado $s \in S \equiv \{Vacío, Planta, Herbívoro, Carnívoro\}$. El autómata se actualiza de la siguiente manera (suponiendo n iteraciones):

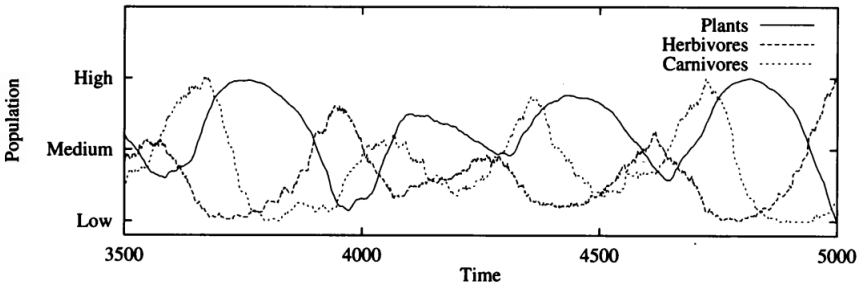

Figura 3.4: Niveles de población de todas las especies normalizadas. Fuente: [4]

[3]Véase la sección 3.2 de la memoria del TFG de Matemáticas.

[4][8]

```
1: for 0:n do
2:     for cada celda Vacío, e do
3:         if e tiene 3 o más vecinos son Planta then
4:             El estado de e cambia a Planta
5:         end if
6:     end for
7:     for cada celda Herbívoro, h (en orden aleatorio) do
8:         Decrementar la reserva de energía de h una cantidad fija.
9:         if La reserva de energía de h se agota then
10:            El estado de h cambia a Vacío
11:        else if Hay al menos un vecino v que es Planta then
12:            h se mueve a la posición de v y consume la energía de v (la posición inicial quedará con el estado Vacío y
    la final con el estado Herbívoro)
13:            if h tiene suficiente energía then
14:                La posición inicial de h pasa a tener estado Herbívoro (se reproduce)
15:            end if
16:        else
17:            h se mueve a una celda vecina aleatoria que esté en el estado Vacío
18:        end if
19:    end for
20:    for cada celda Carnívoro, c (en orden aleatorio) do
21:        Decrementar la reserva de energía de c una cantidad fija.
22:        if La reserva de energía de c se agota then
23:            El estado de c cambia a Vacío
24:        else if Hay al menos un vecino h que es Herbívoro then
25:            c se mueve a la posición de h y consume la energía de c (la posición inicial quedará con el estado Vacío y
    la final con el estado Carnívoro)
26:            if h tiene suficiente energía then
27:                La posición inicial de h pasa a tener estado Herbíboro (se reproduce)
28:            end if
29:        else
30:            c se mueve a una celda vecina aleatoria, con prioridad para los vecinos con estado Vacío
31:        end if
32:    end for
33: end for
```

Tabla 3.1: Pseudocódigo ejemplo de autómatas celulares

Si ejecutamos el programa en una cuadrícula de dimensiones 100x100 y 5000 iteraciones obtenemos los resultados plasmados en la figura 3.4. Se observa que los picos en la población de plantas son seguidos por picos en la población de herbívoros que a su vez provocan picos en la población de carnívoros.

3.4 Motores de simulación

La creciente popularidad de los videojuegos ha hecho que sus motores gráficos hayan evolucionado hasta ser mucho más accesibles y flexibles para el usuario medio. Hoy en día prácticamente cualquier motor de videojuegos tiene herramientas básicas implementadas [9] de base como un sistema de físicas, control del input del usuario, renderizado de imágenes y modelos 3D, detección de colisión, inteligencia artificial o scripting. Los tres más conocidos son Unreal Engine, Unity y GameMaker aunque este último es menos potente y está más orientado a usuarios sin experiencia en programación.

Estas características los hacen perfectos para crear simulaciones, por ejemplo, se ha usado Unreal Engine para crear entrenamientos en base a simulación en campos como la defensa o la exploración espacial. Incluso se ha estudiado crear un estándar [10] para promover la modularización de las arquitecturas de videojuegos en pos de facilitar la creación de modelos y simulaciones de defensa militar. El equipo de trabajo de la OTAN concluyó que las tecnologías de videojuegos comerciales combinados con los estándares apropiados pueden mejorar la eficiencia, calidad, flexibilidad y reusabilidad de componentes en sistemas de simulación para reducir los costes de creación de los mismos aunque sería necesario estándares flexibles para evitar cambios bruscos en las tecnologías. Esto no es posible por el momento ya que los motores de videojuegos son en su mayor parte cerrados, no permiten la inclusión de librerías de terceros y trasladar componentes de un motor a otro es costoso. No obstante, estos motores se han usado en distintos estudios [11] de diversas maneras, por ejemplo, en la Universidad de Michigan utilizaron el motor Unreal para crear un espacio de simulación donde una inteligencia artificial fue puesta a prueba mientras controlaba a un personaje dentro de un videojuego.

Existen cientos de motores de simulación, se pueden encontrar muchos de ellos en la página de *The European Multidisciplinary Society for Modelling and Simulation Technology* (Eurosis). Existen desde simulaciones basadas en agentes como *Lapsang* o *MASON* que funcionan en unidades de tiempo discretas, hasta simulaciones que utilizan *machine-learning* para simular la distribución de especies alrededor del mundo según condiciones climáticas como hace *Maxent*.

Capítulo 4

Contexto de desarrollo

En este capítulo se presentarán la metodología de trabajo empleada y las herramientas utilizadas en el desarrollo de la aplicación.

4.1 Metodología

El enfoque utilizado durante el desarrollo de la aplicación ha sido el iterativo-incremental. Este tipo de desarrollo [12] comenzó a usarse en la década de 1960 en la NASA con el Proyecto Mercury, un programa de vuelos espaciales tripulados de Estados Unidos en el que construyeron la nave espacial Mercury cuyo diseño fue modificado cuatro veces a lo largo de 2 años durante los cuales se iban realizando experimentos para refinar el mismo.

Se ha seguido un refinamiento del desarrollo iterativo-incremental, que es el desarrollo en espiral el cual sigue el ciclo que se muestra en la figura 4.1. Una iteración se divide en las siguientes fases:

1. **Determinar objetivos**: En una primera iteración se obtienen unos objetivos a partir de un concepto inicial del producto, estos objetivos permiten tener un prototipo al final de la iteración. En las siguientes iteraciones los objetivos se obtienen del producto resultante de la anterior etapa.

2. **Identificar y resolver riesgos**: Se identifican cuáles son los problemas que pueden surgir en la implementación por conflictos entre requisitos u otros motivos. En este

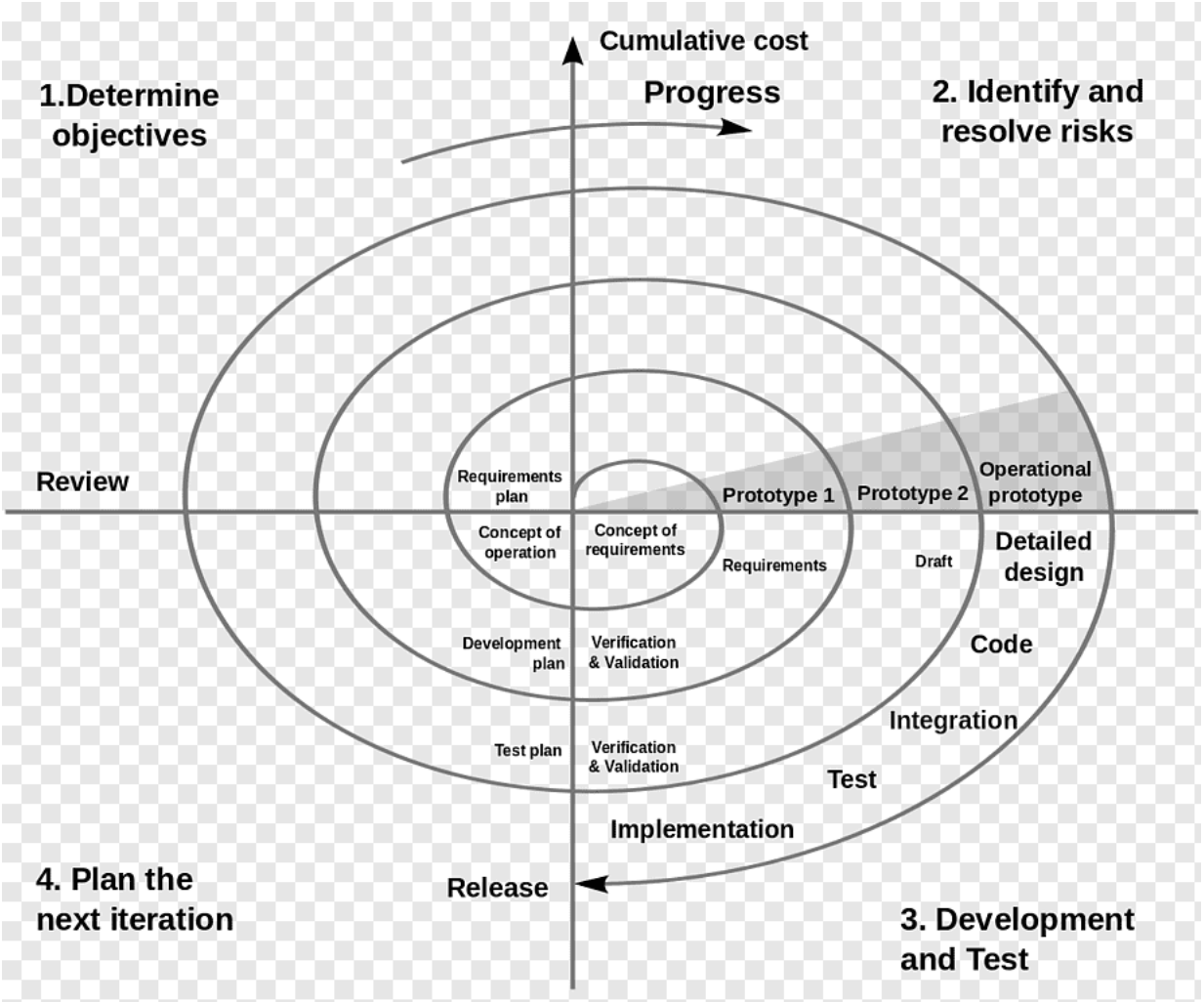

Figura 4.1: Ciclo iterativo incremental

proyecto la fase de identificación de riesgos se sustituye por la de diseño en base a objetivos aunque también se realiza una evaluación de calidad del código en base a los principios SOLID.

3. **Desarrollo y Testeo**: En esta fase se implementa el diseño de la fase 2 y se hacen pruebas con el código para detectar los puntos débiles del producto.

4. **Planificación**: Una vez detectados los puntos débiles, se proponen cambios en el diseño y se plantean nuevos objetivos que acerquen el prototipo al producto final deseado.

Para el desarrolo de esta aplicación, en cada iteración se comienza planteando unos objetivos, a partir de estos se valora un diseño que permita cumplirlos a la vez que generar

un código flexible y mantenible, todo ello documentado en este TFG con diagramas UML detallados. Una vez implementados los cambios en la aplicación, se hacen experimentos con los que analizar el estado de la simulación y de los que extraer cambios para la misma. Además se comprueba el cumplimiento de ciertos principios de programación con los que detectar posibles refactorizaciones que implementar en la siguiente etapa de desarrollo.
Todo el proceso de desarrollo queda registrado en el siguiente repositorio de github: `https://github.com/Dacarpe03/Ecosystem-Simulator`.

4.2 Motor gráfico Unity

Para visualizar la simulación se utilizará Unity, un motor gráfico creado en 2005 por Unity Technologies que cuenta con un entorno de desarrollo para C#.
Un motor gráfico [13] es un software que usan otras aplicaciones o programas para dibujar gráficos en las pantallas de ordenador.
Unity usa un enfoque basado en componentes o *Prefabs* [14], objetos que cuentan con componentes gráficos y scripts que definen su comportamiento y hacen de plantilla para sus instancias. Dichas instancias se colocan en *Scenes* [15], un fichero que contiene el entorno y/o menús de la aplicación, se puede pensar en una *Scene* en términos de un nivel de videojuegos.
Más de 5 millones de personas usan a día de hoy este motor, lo que ha construido una comunidad que comparte consejos, librerías e incluso *Prefabs* que facilitan del desarrollo en esta herramienta. Entre los usos [16] que se le da encontramos:

- **Desarrollo de videojuegos**: Gracias a su gran comunidad, crear un videojuego en Unity se ha vuelto menos complicado. Los desarrolladores también eligen Unity porque tiene herramientas que facilitan la monetización de juegos móviles.

- **Experiencias interactivas**: La aparición de las pantallas táctiles está propiciando la transición a aplicaciones más interactivas e intuitivas para el usuario, por ejemplo, los mapas de los centros comerciales ahora son táctiles. Es fácil crear este tipo de aplicaciones en Unity ya que permite la portabilidad entre una gran variedad de dispositivos.

- **Storyboarding**: En la *Asset Store* podemos encontrar modelos simples de personajes para usarlos en una *Scene* de Unity y presentar la idea de una película de una forma más visual.

- **Animación**: Unity también permite crear animaciones 2D o 3D.

- **Arquitectura**: Unity puede soportar una gran cantidad de renderizado y formas geométricas complejas. Es relativamente fácil importar ficheros desde *Sketchup*[1] a Unity y editar una *Scene* que permita visualizar el edificio a construir.

- **Simulación**: Unity también ha comenzado a usarse para simulaciones de procesos industriales, simulaciones para entrenamientos de pilotos o incluso de médicos como es el caso de *Clinispace Virtual Sim Center*.

Para el sistema desarrollado se aprovechará Unity para crear modelos 3D de animales y visualizar su posición durante la simulación.

4.3 Lenguaje de programación C#

C# [17] es un lenguaje de programación de propóstio general, moderno, de código abierto y orientado a objetos. Fue desarrollado por Anders Hejlsberg y su equipo dentro de la iniciativa .Net y fue aprobado por la European Computer Manufacturers Association (ECMA) y la International Standards Organization (ISO). C# se encuentra entre los lenguajes de Common Language Infrastructure (CLI) [2]. Sintácticamente es muy parecido a Java y es fácil de usar para usuarios familiarizados con C, C++ o Java.

C# es uno de los lenguajes del framework .Net, es decir, las aplicaciones desarrolladas en este framework son aplicaciones multiplataforma.

[1] Herramienta de modelado 3D con sistema de iluminación dinámica usada principalmente por arquitectos.

[2] Especificación abierta que describe código ejecutable y un entorno de ejecución que permite a los lenguajes de alto nivel ser usados en distintas plataformas sin necesidad de ser reescritos para arquitecturas específicas.

[3] `https://www.tiobe.com/tiobe-index/`

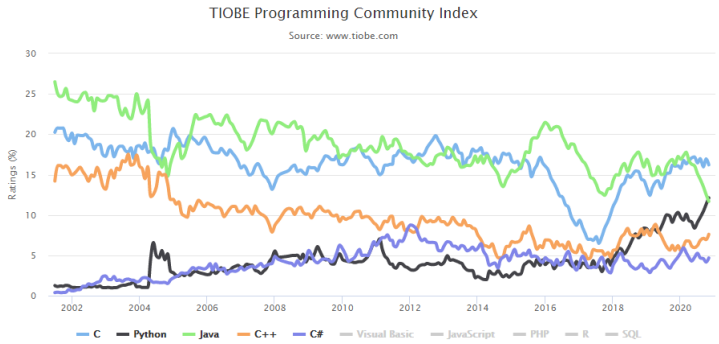

Figura 4.2: Porcentaje de usuarios por lenguaje de programación. Fuente: [3]

Actualmente, C# es el quinto lenguaje de programación más usado [18], siendo los más usados C, Python, Java, C++, en ese orden como podemos ver en la figura 4.2.

C# cuenta con propiedades. Una propiedad[19] es un miembro que proporciona un mecanismo flexible para leer, escribir o calcular el valor de un campo privado. Las propiedades se pueden usar como si fueran miembros de datos públicos, pero en realidad son métodos especiales denominados descriptores de acceso. Esto permite acceder fácilmente a los datos a la vez que proporciona la seguridad y la flexibilidad de los métodos.

Además de las propiedades, C# permite usar consultas LINQ[20]: Language-Integrated Query (LINQ) es el nombre de un conjunto de tecnologías basadas en la integración de capacidades de consulta directamente en el lenguaje C#. Tradicionalmente, las consultas con datos se expresaban como cadenas simples sin comprobación de tipos en tiempo de compilación ni compatibilidad con IntelliSense. Además, tendrá que aprender un lenguaje de consulta diferente para cada tipo de origen de datos: bases de datos SQL, documentos XML, varios servicios web y así sucesivamente. Con LINQ una consulta es una construcción de lenguaje de primera clase, como clases, métodos y eventos. Escribe consultas en colecciones de objetos fuertemente tipadas con palabras clave del lenguaje y operadores familiares. La familia de tecnologías de LINQ proporciona una experiencia de consulta coherente para objetos (LINQ to Objects), bases de datos relacionales (LINQ to SQL) y XML (LINQ to XML).

El modelo lógico de la simulación será implementado en C#.

Capítulo 5

Funcionamiento de la simulación

En este capítulo se propone el posible funcionamiento final de la simulación. Puede que el diseño cambie a lo largo del desarrollo de la aplicación. Simplemente se plasma la idea inicial de la que surje este TFG.

5.1 Flujo de la simulación

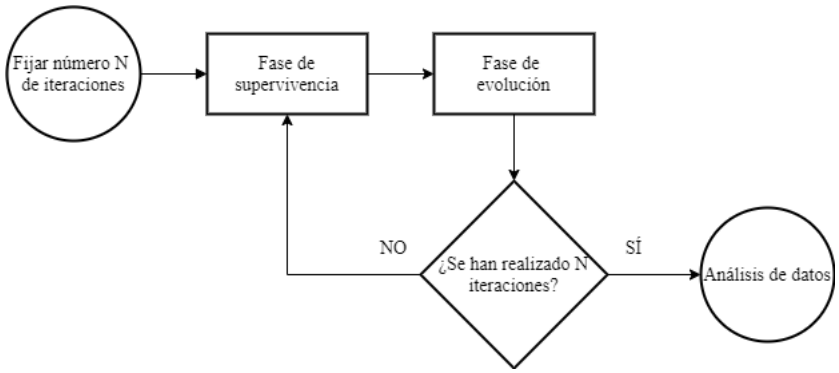

Figura 5.1: Diagrama de flujo de la simulación

Como se indica en la Figura 5.1, al inicio de la simulación se define un número N de iteraciones que se dividen en dos fases, la fase de supervivencia y la fase de evolución. Al finalizar las iteraciones se analizarán los datos obtenidos, presentando los resultados en gráficos.

5.2 Fase de supervivencia

Al comienzo de esta fase los animales se crearán en posiciones aleatorias y una vez inicializados intentarán conseguir un objetivo para sobrevivir que variará según la especie a la que pertenezca.

- **Presas**: Sobrevivirán si cumplen una de estas dos condiciones:

  a) Han huido hasta una una zona segura.

  b) Se han mantenido con vida hasta que todos los predadores han cazado.

- **Predadores**: Sobrevivirán si consiguen cazar [1] una presa.

5.3 Fase de evolución

5.3.1 Ciclo de vida

- **Presas**:

  1. Una cría se convierte en adulto si ha sobrevivido un número determinado de iteraciones.
  2. Un adulto tiene mayor probabilidad de reproducirse cuantas más iteraciones haya sobrevivido.
  3. Un adulto muere al cabo de un número determinado de iteraciones si no es cazado antes.

- **Predadores**:

  1. Una cría se convierte en adulto si consigue cazar en menos de un número determinado de iteraciones.
  2. Un adulto tiene mayor probabilidad de reproducirse cuantas más iteraciones haya sobrevivido.
  3. Un adulto muere si no ha conseguido cazar en un número determinado de iteraciones consecutivas.

[1] Las estrategias de huida y de caza se detallarán en el TFG de Matemáticas con título "*Simulación de estrategias de caza de animales gregarios a partir de metaheurísticas*", capítulo 6.

5.3.2 Reproducción

Cada individuo tiene un pseudo-ADN en forma de una cadena de caracteres con distintas letras que definirán las características del animal. Por ejemplo: VVVTTEEEEEE (V para velocidad, T para tamaño, E para esperanza de vida) significará que el individuo tiene velocidad 3, tamaño 2 y esperanza de vida 6.

Cuando dos adultos se reproduzcan y tengan una cría, el ADN de esta se creará mediante un algoritmo genético que combinará los ADNs de sus progenitores.

Capítulo 6

Principios SOLID

Los principios SOLID [21] fueron definidos por Robert C. Martin en un ensayo con nombre *Desing Principles and Design Patterns*. Estos principios tienen como objetivo reducir el acoplamiento y aumentar la cohesión del código. Son clásicos en el diseño orientado a objetos. SOLID es un acrónimo donde cada letra representa un principio distinto:

- ***S****ingle Responsibility Principle* o Principio de Responsabilidad Única.
- ***O****pen Close Principle* o Principio Abierto-Cerrado.
- ***L****iskov Substitution Principle* o Principio de Sustitución de Liskov.
- ***I****nterface Segregation Principle* o Principio de Segregación de Interfaces.
- ***D****ependency Inversion Principle* o Principio de Inversión de Dependencias.

Estos principios se usan normalmente a bajo nivel, pero aquí se extrapolarán como herramienta de análisis y diseño, para lograr los grados de cohesión y bajo acoplamiento deseables.

6.1 Principio de Responsabilidad Única

Toda clase debería tener un único motivo para cambiar. Cada componente del sistema tiene una única función asignada dentro de una tarea más grande. Si la clase tiene más de un motivo para cambiar significa que se encarga de más de una función y por lo tanto debería dividirse en más clases.

6.2 Principio Abierto-Cerrado

Las entidades software deben estar abiertas para su extensión pero cerradas para su modificación. Abiertas para su extensión en cuanto a extender su comportamiento y cerradas en modificación en cuanto a que su código debe ser inalterable. Un módulo ha de ser escrito de forma que pueda extenderse sin que sea necesaria su modificación. Se consigue usando abstracción y polimorfismo.

6.3 Principio de Sustitución de Liskov

La herencia ha de garantizar que cualquier propiedad que sea cierta para los objetos supertipo también lo sea para los objetos subtipo. Las clases derivadas deben ser utilizables a través de la interfaz de la clase base sin necesidad de que el usuario conozca la diferencia, es decir, estas subclases deben ser sustituibles por sus clases base.

6.4 Principio de Segregación de Interfaces

Es conveniente disponer de muchas interfaces de cliente específicos que una sola interfaz de propósito general. Los clientes no deben ser forzados a depender de interfaces que no utilizan.

6.5 Principio de Inversión de Dependencias

Los módulos de alto nivel no deben depender de los módulos de bajo nivel, ambos deben depender de las abstracciones. Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.
Mientras que el Principio Abierto-Cerrado establece el objetivo, el Principio de Inversión de Dependencias establece el mecanismo. A partir de abstracciones se establecen puntos de conexión sobre los que el diseño puede ser aplicado o extendido sin necesidad de modificarlo.

Capítulo 7

Patrones de diseño

Tras haber planteado la idea inicial de la simulación y teniendo en mente los principios SOLID, en este capítulo se realizará una investigación sobre los patrones de diseño que se pueden aplicar para implementar el sistema. Todos los patrones que aparecen en este capítulo han sido usados, o al menos considerados, en las sucesivas iteraciones de la arquitectura. Se desarrolla brevemente el funcionamiento de cada uno de ellos para no tener que explicarlos en el contexto de las implementaciones de los capítulos posteriores. Los diagramas UML presentados son la forma usual de implementar estos patrones, sus diseños serán adaptados para el correcto funcionamiento de los mismos dentro del sistema.

7.1 ¿Qué son los patrones de diseño?

Los patrones de diseño [22] o *design patterns*, son una solución general, reutilizable y aplicable a diferentes problemas de diseño de software. La ventaja de usar los patrones es que el código resultante es mantenible, flexible y sólido.

Surgieron, o al menos se documentaron por primera vez en 1994, cuando cuatro autores llamados Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, a los que posteriormente apodarían como *Gang of Four (GoF)*, publicaron un libro titulado *Design Patterns* con 23 patrones de diseño pertenecientes a 3 tipos distintos: creacionales, estructurales y de comportamiento.

7.2 Patrones de creación

Proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización de código existente [23]. Algunos patrones que pueden resultar útiles al diseñar el sistema de simulación se describen a continuación.

7.2.1 Patrón Builder

Permite construir objetos complejos paso a paso, representando distintos tipos y representaciones de un objeto empleando el mismo código de construcción. Como ejemplo podemos tomar la construcción de una casa. Esta puede tener garaje, piscina, jardín o ninguna de estos extras. Una solución podría ser crear clases que hereden de una superclase Casa pero el patrón Builder propone crear una serie de constructores que creen instancias de la clase Casa pero con distintos accesorios. En la figura 7.1 se muestra el diagrama UML típico del patrón.

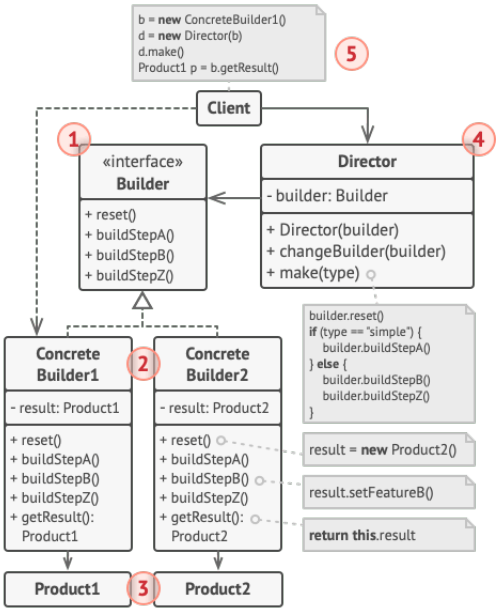

Figura 7.1: Diagrama UML Builder. Fuente: [24]

- **Builder** declara pasos de construcción de un **Producto** que todos los tipos de objetos constructores tienen en común.

- **ConcreteBuilder** son las diferentes implementaciones de los pasos definidos.

- **Director** define el orden en el que se invocarán los pasos de construcción, por lo que puedes crear y reutilizar configuraciones específicas de los productos.

Este patrón podría ser útil a la hora de crear a las distintas especies de animales.

7.2.2 Patrón Factory Method

Es un patrón de diseño que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán [25]. Lo hace de la manera en que se ilustra en la figura 7.2.

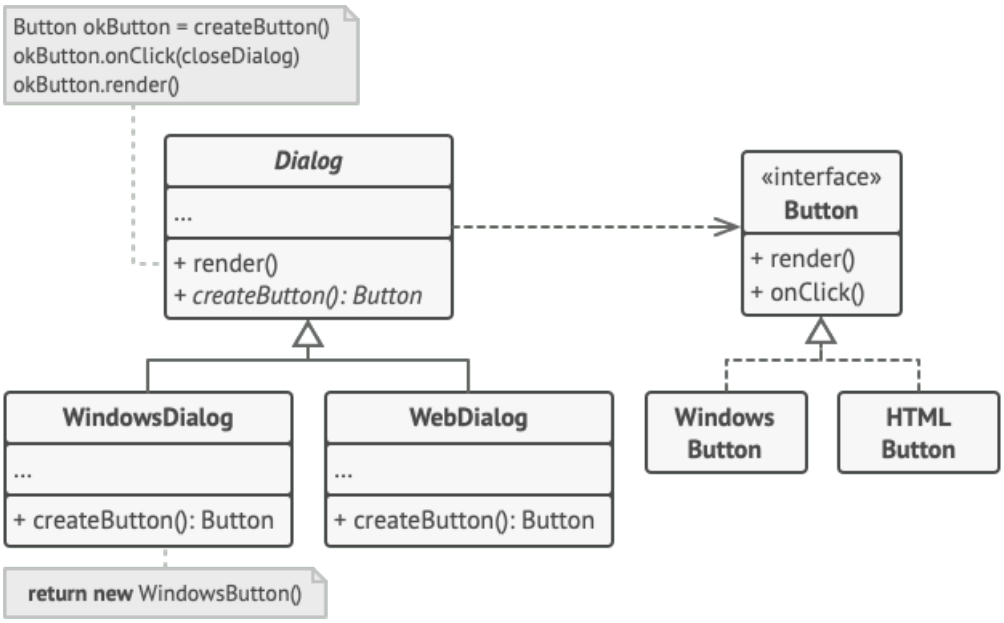

Figura 7.2: Diagrama UML Factory Method. Fuente: [25]

- **Product** declara la interfaz que es común a todos los objetos que puede crear **Creator**.
- **ConcreteProduct** es una implementación de **Product**
- **Creator** declara el método fábrica que devuelve nuevos objetos de producto.
- **ConcreteCreator** son las implementaciones concretas de **Creator**

Este patrón podría ser útil a la hora de crear a las distintas especies de animales.

7.3 Patrones estructurales

Explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de todo el sistema [23]. Algunos patrones que pueden resultar útiles al diseñar el sistema de simulación se describen a continuación.

7.3.1 Patrón Modelo-Vista-Controlador

Este patrón [26] tiene como propósito separar lógica de interfaz y facilitar la evolución por separado de ambos aspectos. Al ser un patrón arquitectónico cada elemento del patrón puede tener su propio diagrama UML por lo que se presenta la estructura principal en la figura 7.3.

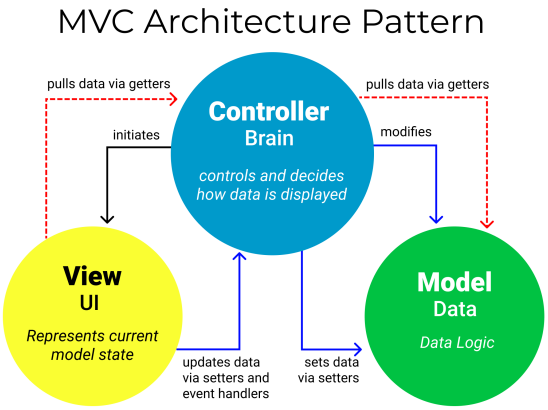

Figura 7.3: Estructura del patrón Modelo-Vista-Controlador. Fuente: [27]

Los tres componentes del patrón son:

- **Modelo**: Contiene la lógica de la aplicación y debe ser independiente del **Controlador** y la **Vista**. Recibe peticiones del controlador y devuelve las salidas oportunas.
- **Controlador**: Recibe entradas en forma de eventos que son traducidos a peticiones a la **Vista** o al **Modelo**.
- **Vista**: Muestra visualmente la información del modelo y permite al usuario hacer peticiones al **Modelo** de manera indirecta a través del **Controlador**.

Gracias a este patrón se desacoplan las distintas partes de un sistema de manera que sean intercambiables, además permite la reutilización de los mismos. Se utilizará este patrón para separar las responsabilidades del sistema de manera eficiente.

7.3.2 Patrón Composite

Este patrón permite fusionar objetos en una jerarquía de árbol de tal manera que se trabaje con esa fusión como si fuera un objeto individual [28].

Supongamos que queremos modelar un sistema de cajas que puede contener cajas más pequeñas y estas a su vez contienen productos Para calcular el precio de una caja podríamos abrir todas las cajas que contiene y valorar los productos que contienen. Un método para hacer esto eficiente es hacer que cada caja tenga un valor calculado previamente.

La estructura usual del patrón es la siguiente:

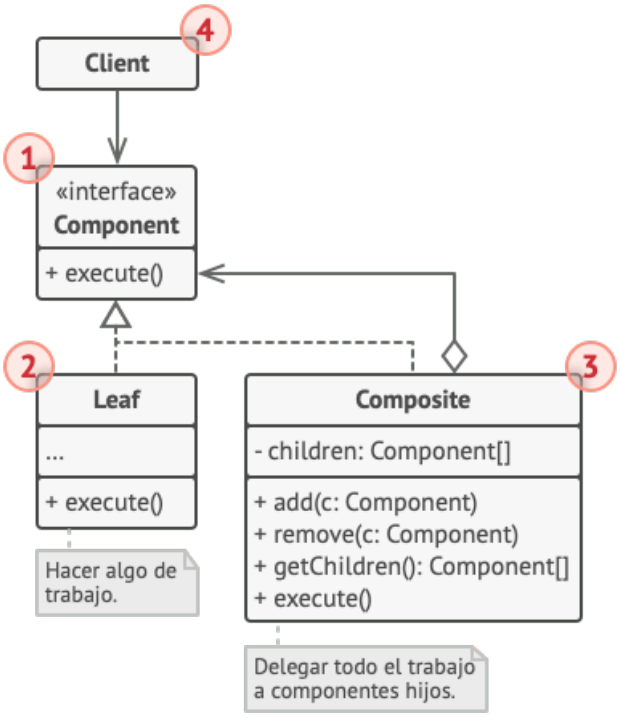

Figura 7.4: Diagrama UML Composite. Fuente: [28]

- **Component** es una interfaz que define las operaciones comunes a cualquier elemento simple o complejo del árbol.

- **Leaf** representa el elemento base de un árbol, no puede tener subelementos.
- **Composite** es un elemento que tiene subelementos: hojas u otros composite. Al recibir una solicitud la delega a sus subcomponentes.
- **Cliente** funciona con todos los elementos a través de la interfaz componente. Como resultado, el cliente puede funcionar de la misma manera tanto con elementos simples como complejos del árbol.

Este patrón podría ser útil a la hora de crear los grupos de animales.

7.4 Patrones de comportamiento

Se encargan de que la comunicación y asignación de responsabilidades entre objetos sea efectiva [23]. Algunos patrones que pueden resultar útiles al diseñar el sistema de simulación se describen a continuación.

7.4.1 Patrón State

Permite que un objeto cambie su comportamiento dependiendo de su estado interno para darle mayor flexibilidad en cuanto a funcionalidad [29]. Es el patrón perfecto para crear un máquina de estados.

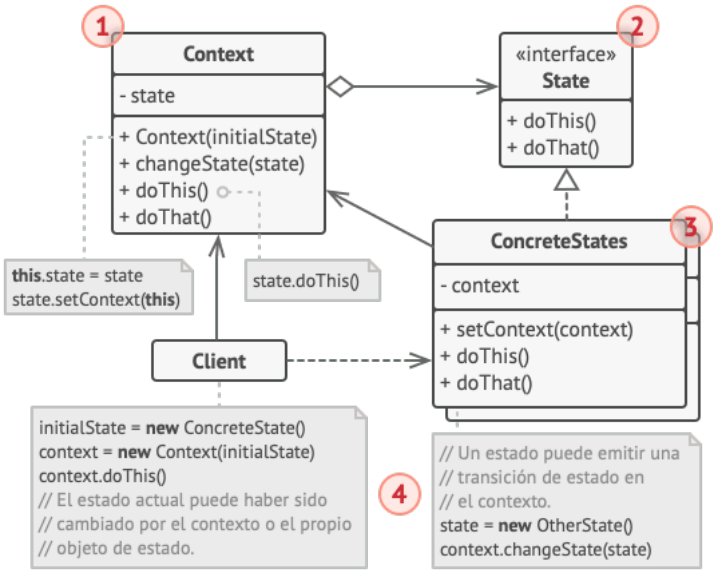

Figura 7.5: Diagrama UML State. Fuente: [30]

- **Context** almacena una referencia a un **ConcreteState** y le delega el trabajo.
- **State** es la interfaz que declara los métodos que debe tener un estado..
- **ConcreteState** es una implementación específica de **State**.

Este patrón serviría para modelar las distintas fases de la simulación.

7.4.2 Patrón Strategy

Define una familia de algoritmos de tal manera que puedan ser implementados de distintas maneras en clases intercambiables entre sí [31]. Se podría usar en una aplicación que crea rutas, con una estrategia distinta para coches, bicicletas y peatones.

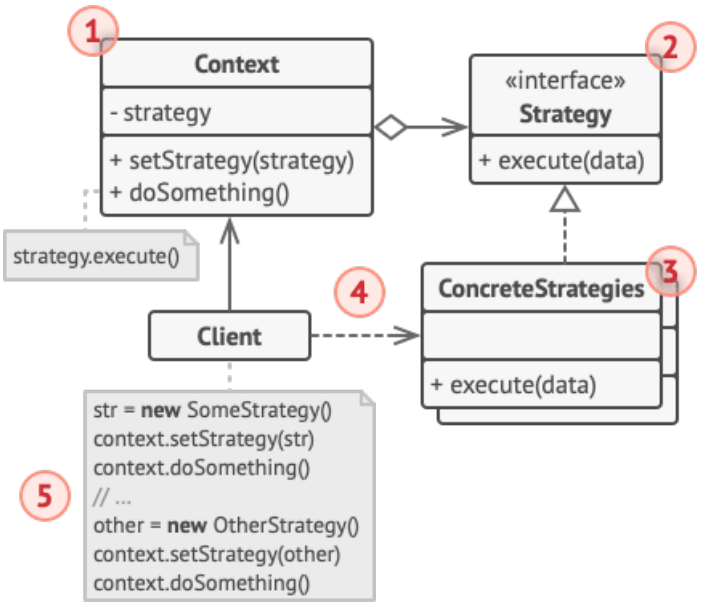

Figura 7.6: Diagrama UML Strategy. Fuente: [31]

Las responsabilidades de las clases presentadas en el diagrama UML 7.6 son las siguientes:

- **Context** contiene una referencia a una estrategia concreta.
- **Strategy** es la interfaz que declara los métodos que debe tener una estrategia.
- **Concretestrategy** es una implementación específica de **Strategy**.

La diferencia entre este patrón y el patrón **State** es que los estados concretos se conocen unos a otros, mientras que las estrategias concretas son completamente independientes. Se podría utilizar este patrón para definir distintas estrategias de caza.

7.4.3 Patrón Mediator

Reduce las dependencias caóticas entre objetos. Restringe las comunicaciones directas entre objetos, forzándolos a colaborar únicamente a través de un objeto mediador [32]. Una analogía con el mundo real sería una torre de control aérea. Los pilotos no hablan entre sí para decidir quién aterriza primero, esta decisión se toma desde la torre de control. En el modelo UML 7.7 se muestra la abstracción del patrón.

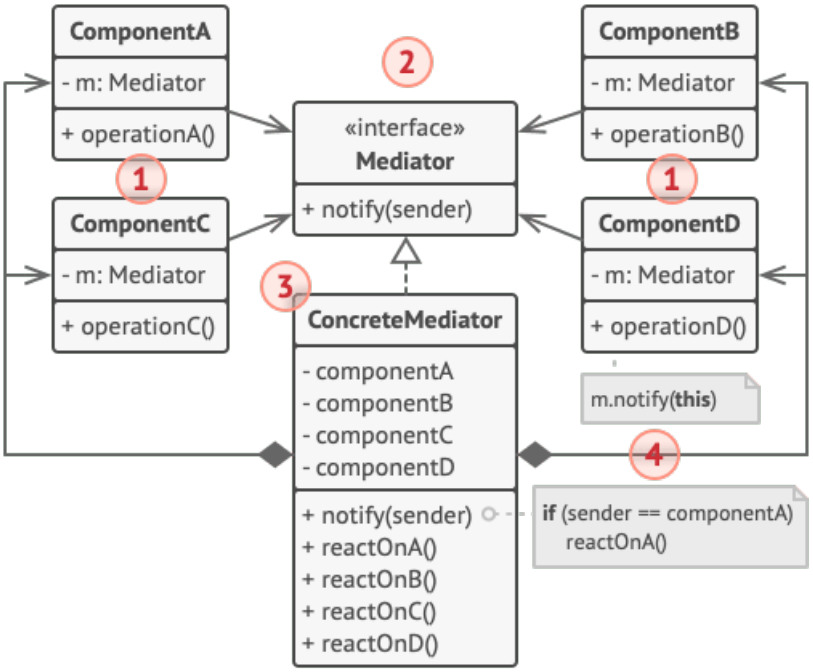

Figura 7.7: Diagrama UML Mediator. Fuente: [32]

- **Components** son las clases que se quieren desacoplar para que se comuniquen de manera indirecta a través del **ConcreteMediator**.
- **Mediator** es la interfaz que declara los métodos de comunicación entre los objetos.
- **ConcreteMediator** encapsulan las relaciones entre varios componentes. Los mediadores concretos a menudo mantienen referencias a todos los componentes que

gestionan incluso a su ciclo de vida.

Este patrón podría ser útil para controlar la comunicación entre depredadores.

7.5 Patrones de optimización

Este tipo de patrones no fueron creados por el *GoF*, sino que se encuentran definidos por Nystrom en el libro *Game Programming Patterns*. Como su nombre indica, optimizan la *performance* del sistema principalmente usando estrategias que lo dividen en subsistemas que tienen las mismas propiedades que el original acelerando la computación de cálculos.

7.5.1 Patrón Spatial Partition

Para un conjunto de objetos, cada uno con una posición en el espacio, se propone almacernarlos en una estructura de datos espacial que organice los objetos según su posición. Esta estructura de datos permite obtener objetos que estén cercanos entre ellos de una manera eficiente. Cuando un objeto actualiza su posición, también se actualiza la estructura de datos [30].

La estructura del patrón es la siguiente:

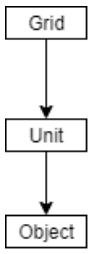

Figura 7.8: Diagrama UML Spatial Partition. Fuente: [30]

- **Grid** es el espacio por el que se colocan los objetos. Está formado por **Unit**. En el caso de la simulación sería el ecosistema.
- **Unit** representa una unidad del espacio, en esta clase se encuentran los objetos del sistema.
- **Object** es la clase que se mueve por el espacio. En el caso de la simulación sería un animal.

Más que un patrón de diseño, es un patrón de programación aunque para esta aplicación se considera al mismo nivel ya que en la simulación la posición de un animal es una entidad de primera clase y no una simple propiedad. Es por esto que se considera este patrón para acelerar la ejecución de la simulación; dividiendo el espacio en partes iguales se computan las interacciones entre animales que se encuentren en una misma unidad de espacio reduciendo así las interacciones totales que podría haber en todo el ecosistema.

Capítulo 8

Etapa 0: Planteamiento inicial

Esta primera etapa es una fase preliminar del desarrollo, por eso se denomina Etapa 0. En ella se plasma el diseño inicial del sistema en un diagrama UML. Se comienza planteando los objetivos de la etapa. A continuación, se diseña el modelo UML para el sistema en base a esos objetivos.

8.1 Objetivo

El principal objetivo será crear un diagrama UML que modele la arquitectura del sistema al que se pretende llegar. Nace de la idea inicial y se usa como guía para el comienzo del desarrollo, por lo que es probable que el resultado final sea distinto al diagrama UML del sistema final.
Se puede consultar el UML completo en el apéndice A.1

8.2 Diseño

La arquitectura principal del sistema elegida es Modelo-Vista-Controlador ya que se utilizará el motor gráfico Unity para visualizar la simulación. La idea es que Unity reciba directamente las posiciones de los animales y los represente sin necesidad de realizar los cálculos necesarios para simular sus movimientos e interacciones. El controlador se usará para iniciar la aplicación y pasar la información del modelo a la vista. La vista es la única parte del sistema que depende de Unity porque el modelo puede ejecutarse como una

aplicación de escritorio al haberse implementado de manera independiente a los demás componentes de la arquitectura. En el modelo se realizan todas las operaciones y cálculos con los que funciona la simulación.
Separar las responsabilidades de esta manera facilita crear un diseño más eficiente para cada una de las partes del sistema como se explica a continuación.

8.2.1 Modelo

En la figura 8.1 se muestra el UML del modelo que está formado por distintas clases:

- **Context**: Hace de ecosistema en el que existen dos grupos de animales, uno de presas y otro de predadores.
- **Group**: Clase abstracta que representa un grupo de animales con una determinada jerarquía, es abstracta porque grupos de distintas especies se comportan de manera distinta pero comparten funciones como comer o reproducirse.
- **Herd**: Clase que hereda de Group y representa a un grupo de presas.
- **Prey**: Clase que hereda de Group y representa a un grupo de predadores.
- **Hierarchy**: Clase que le da un orden a un conjunto de animales
- **Animal**: Clase abstracta que representa un animal, es abstracta porque distintas especies de animales realizan las mismas funciones de forma distinta pero comparten ciertos atributos.
- **Breeding**: Representa una cría de animal y hereda de Animal.
- **Adult**: Representa un animal adulto y hereda de Breeding.
- **Dna**: Clase que guarda la información de un animal, su pseudo-ADN.
- **State**: Interfaz que implementa Animal, representa el estado en el que se encuentra.
- **Flee**: Representa el estado de huida.
- **Eat**: Representa el estado de comer.

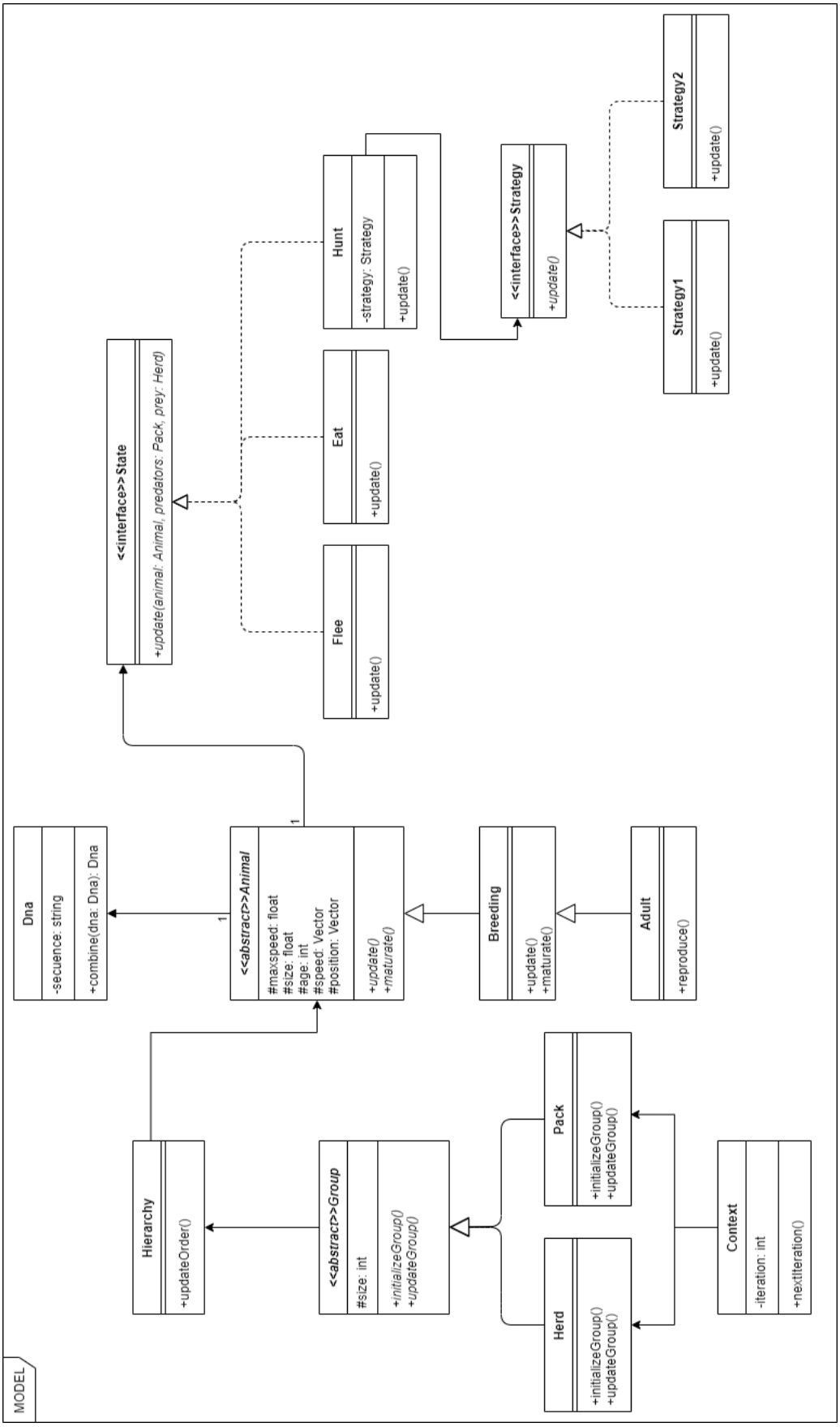

Figura 8.1: Diagrama UML del modelo

- **Hunt**: Representa el estado de caza e implementa Strategy
- **Strategy**: Interfaz con la que se podrán implementar distintas estrategias de caza.

En la figura 8.1 también se aprecia el uso de dos patrones de diseño:

- **Patrón State**: Se usa este patrón porque permite modelar a los animales de tal manera que sean máquinas de Turing, ya que podrán transicionar entre distintos estados.
- **Patrón Strategy**: Este patrón permitirá implementar distintas estrategias de caza para usar la más conveniente en cada experimento.

8.2.2 Vista

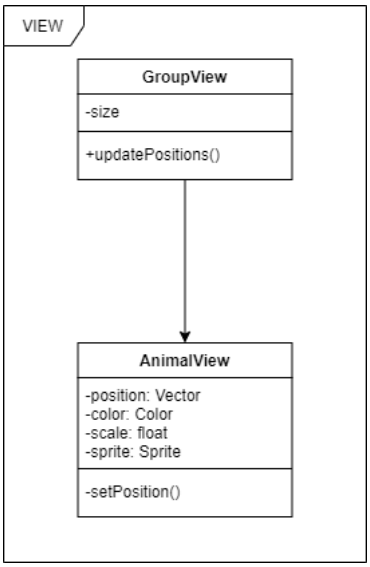

Figura 8.2: Diagrama UML de la vista

En la vista únicamente tendremos dos clases:

- **GroupView**: Clase que almacena una lista de animales.
- **AnimalView**: Clase que tiene un modelo 3D del animal, su color y su posición.

8.2.3 Controlador

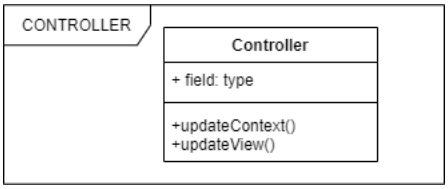

Figura 8.3: Diagrama UML del controlador

En el controlador tenemos la clase Controlador que traspasará datos del modelo a la vista.

Capítulo 9

Etapa 1: Prototipo

Esta segunda estapa es la etapa más larga, ya que se desarrolla la primera iteración del diseño específicado en la etapa 0 que es complejo pero no refinado. Se parte de la nada y se llega a tener un prototipo de simulación completo.
En primer lugar, se definirán los objetivos de la etapa. A continuación se detallará el diseño con ayuda de gráficos UML a la vez que se explican los patrones utilizados. Posteriormente se analizará el cumplimiento de los principios SOLID. Para concluir el capítulo se realizarán experimentos en la simulación para definir los objetivos de la siguiente etapa y detectar refactorizaciones necesarias.

9.1 Objetivos

El objetivo principal es crear una estructura prototipo de la simulación que se puede dividir en subobjetivos:

- Implementar el modelo del sistema.
- Implementar el algoritmo de huida de las presas.
- Implementar un algoritmo *dummy*[1] de caza.
- Implementar el controlador del sistema.
- Implementar la vista del sistema.

[1] Algoritmo simple que hace de *placeholder* para los algoritmos de caza finales

- Realizar una serie de simulaciones variando los parámetros del sistema.
- Conclusiones de la etapa.

9.2 Estado de la simulación

En esta etapa la simulación funciona de la siguiente manera:

1. Inicialización de los parámetros iniciales de la simulación, que son:
    - **ITERATIONS**: Número de iteraciones de la simulación.
    - **PREY_REPRODUCTION_RATE**: Probabilidad de que una pareja de presas tenga una cría.
    - **PREY_VISION_RADIUS**: Radio de visión de las presas.
    - **PREY_MAX_SPEED**: Velocidad máxima que pueden alcanzar las presas.
    - **PREY_GROUP_SIZE**: Tamaño inicial de la población de presas.
    - **PREDATOR_REPRODUCTION_RATE**: Probabilidad de que una pareja de predadores tenga una cría.
    - **PREDATOR_VISION_RADIUS**: Radio de visión de los predadores.
    - **PREDATOR_MAX_SPEED**: Velocidad máxima que pueden alcanzar los predadores.
    - **PREDATOR_GROUP_SIZE**: Tamaño inicial de la población de predadores.
2. Se inicializa el ecosistema en el estado **SimulationSurviveState**
3. Se inicializa un **AnimalGroup** con el tamaño inicial **PREY_GROUP_SIZE** donde todos los animales del grupo tienen el estado inicial **AnimalFleeState** que representa a las presas.
4. Se inicializa un **AnimalGroup** con el tamaño inicial **PREDATOR_GROUP_SIZE** donde todos los animales del grupo tienen el estado inicial **AnimalHuntState** que representa a los predadores.
5. Las presas huyen hacia una zona segura mientras que los predadores intentan cazarlas.

6. Cuando todas las presas que no han sido cazadas han llegado a la zona segura, el ecosistema pasa al estado **SimulationEvolveState**. En este estado los animales se reproducen de la siguiente manera:

   - Todas las presas que hayan alcanzado la zona segura sobreviven, y por cada pareja existente , se intenta crear otra presa según la probabilidad definida en la variable **PREY_REPRODUCTION_RATE**.
   - Todos los predadores que hayan cazado al menos a una presa sobreviven, y por cada pareja existente se intenta crear otro depredador según la probabilidad **PREDATOR_REPRODUCTION_RATE**.

7. Se finaliza la iteración y se vuelve al paso 3 aunque los grupos tendrán un tamaño inicial igual a los supervivientes + nacidos en esta iteración. Si el número de iteraciones que ha hecho la simulación es igual a el parámetro **ITERATIONS**, entonces termina.

La simulación puede terminar antes de realizar el número indicado de iteraciones si uno de los dos grupos de animales se extingue.

Todo el proceso anterior se ilustra en el siguiente vídeo: `https://www.youtube.com/watch?v=pbKpSuOO9mU`

9.3 Diseño

Al final de esta etapa tenemos la arquitectura MVC implementada. El modelo UML completo se puede consultar en el apéndice A.2

9.3.1 Modelo

En primer lugar se expone la clase **Ecosystem** que hace de contenedor para las demás:

Esta clase contiene los dos grupos de animales que interactuarán en la simulación: *preys* y *predators*.

Para su constructor necesitamos ciertos parámetros para los grupos de animales que se

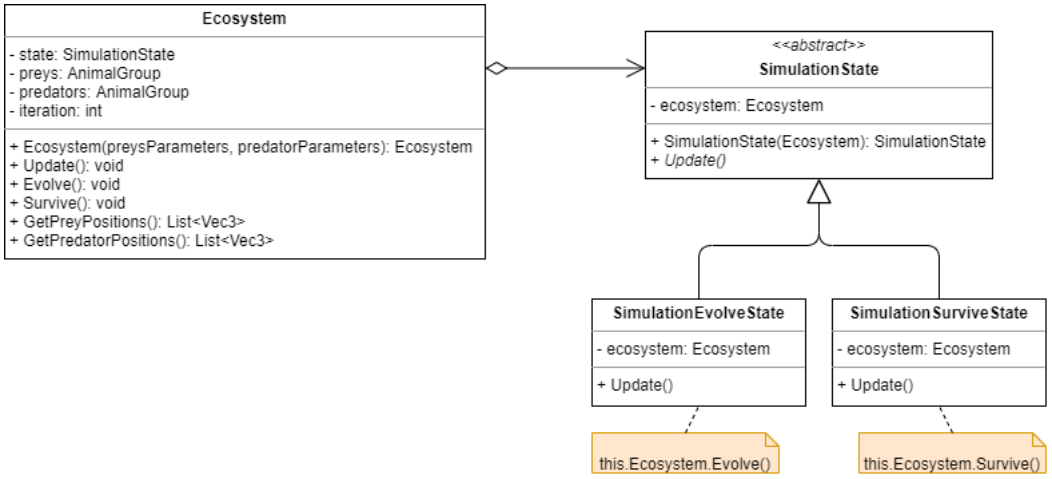

Figura 9.1: Diagrama UML de Ecosystem en la etapa 1

muestran en la figura 9.2. La simulación funciona frame a frame llamando al método el método *Update()* de **Ecosystem** usa que hace que los grupos de animales evolucionen o sobrevivan según el estado de la simulación en que se encuentre.

En el modelo UML también se aprecia la implementación del patrón State a partir de la clase **SimulationState** que modela los distintos estados en los que puede estar la simulación: **SimulationEvolveState**, que representa el estado de evolución en el que los animales se reproducen y evolucionan, y **SimulationSurviveState**, que representa el estado de la simulación en el que los predadores están cazando a las presas que intentan huir. Se usa el patrón State para que se puedan añadir nuevos estados a la simulación con mayor facilidad.

Los métodos *GetPreyPositions()* y *GetPredatorPosition()* serán usados por el Controlador de la aplicación (ver figura 9.7).

Comparando con la figura 8.1 la clase **Ecosystem** ha sustituido a **Context** y se ha añadido un patrón State que fue necesario para el correcto funcionamiento de la simulación.

La siguiente clase a analizar es **AnimalGroup**:

Hace de contenedor de un conjunto de animales que guarda los datos de la especie tales como *reproductionProb, maxSpeed, visionRadius, arePrey*, que representan la probabilidad de reproducirse que tienen dos individuos de la misma especie, la velocidad máxima que puede alcanzar un ejemplar del grupo, su radio de visión y si son presas o no, re-

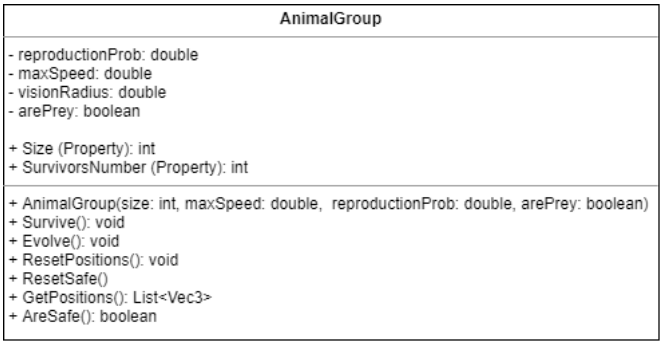

Figura 9.2: Diagrama UML de AnimalGroup en la etapa 1

spectivamente. El constructor necesita estos parámetros para instanciar los animales y dependiendo de si son presas o predadores, les asigna el estado inicial correspondiente. También se observa el uso de propiedades de C# para el tamaño del grupo o *Size*, que varía a lo largo de la simulación y necesita recalcularse cada vez que se usa el *getter* de la propiedad, y para *SurvivorsNumber* que devuelve el resultado de una sentencia LINQ. Los métodos *Survive()* y *Evolve()* se llaman según el estado de la simulación. *ResetPosition()* y *ResetSafe()* son métodos que se usan al iniciar un nuevo estado de supervivencia para reiniciar las posiciones y estados de los animales mientras que *AreSafe()* se usa para determinar el fin de la fase de supervivencia. *GetPositions()* es un método auxiliar que usa el ecosistema para devolver las posiciones de los animales a los grupos al **Controller**. Comparando con la figura 8.1 se observan las siguientes diferencias:

1. **AnimalGroup** no es una clase abstracta.

2. **Herd** y **Pack** no existen, ya que la distinción entre especies se modela mediante los estados de los animales como se ve en la figura 9.3.

3. **AnimalGroup** carece de **Hierarchy** ya que en esta etapa no se ha implementado esa clase.

AnimalGroup tiene una lista de **Animal**, la clase que modela un animal:

La clase **Animal** tiene un constructor con el que se le asigna las características de la especie que se inicializa, incluido un estado inicial. Esta clase también cuenta con dos vectores que representan su posición, *position*, y velocidad, *speed*, en un instante de la simulación. Los métodos relacionados con el movimiento del animal son *Move()*

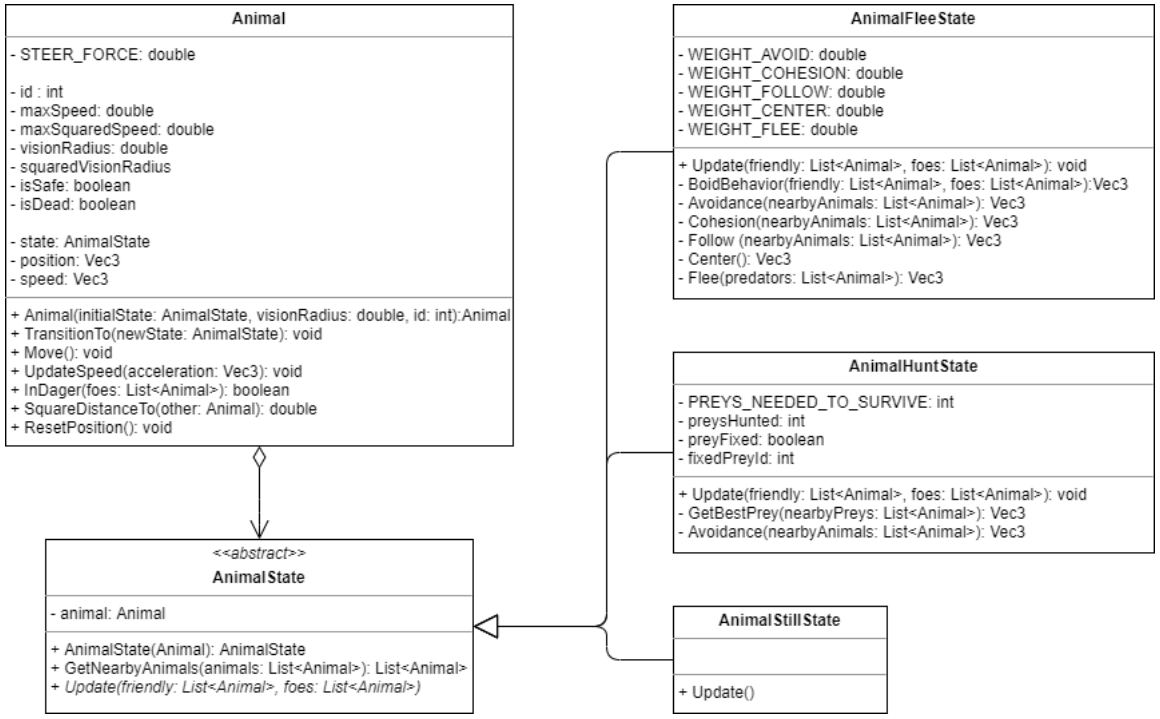

Figura 9.3: Diagrama UML de Animal y AnimalState en la etapa 1

y *UpdateSpeed()*. Los métodos *InDanger(), SquaredDistanceTo(), ResetPosition()* son métodos auxiliares que se usan en los estados.

De nuevo se ha vuelto a usar el patrón State, esta vez porque los animales se han modelado de tal manera que sean autómatas que transicionen entre distintos estados. Los distintos estados en los que se puede encontrar una instancia de **Animal** son:

- **AnimalFleeState**: Es el estado inicial de las presas, el algoritmo que implementa es la estrategia de huida de las mismas. Este algoritmo se detalla en la memoria del TFG de Matemáticas en la sección 6.1.

- **AnimalHuntState**: Es el estado inicial de los predadores. En este punto del desarrollo el algoritmo que implementa es muy simple ya que únicamente hace que un predador persiga a la presa que maximice el resultado de dividir la distancia hasta la presa entre su velocidad. Este algoritmo será sustituido más adelante.

- **AnimalStillState**: Es el estado final de las presas al que transicionan cuando alcanzan una línea de meta a partir de la cual están seguras.

Comparando con la figura 8.1 se observan las siguientes diferencias:

1. No existe la clase **Dna** ya que todavía no se ha implementado.

2. No existen las clases **Breeding** y **Adult** porque la simulación está en un estado en la que no es necesaria esa distinción.

3. El patrón Strategy no está implementado en **AnimalHuntState** porque sólo se ha codificado una estrategia de caza que además está haciendo de *placeholder* para las que se modelen en siguientes etapas del desarrollo.

Por último, en la figura 9.4, se detalla el núcleo de la fase de supervivencia de la simulación.

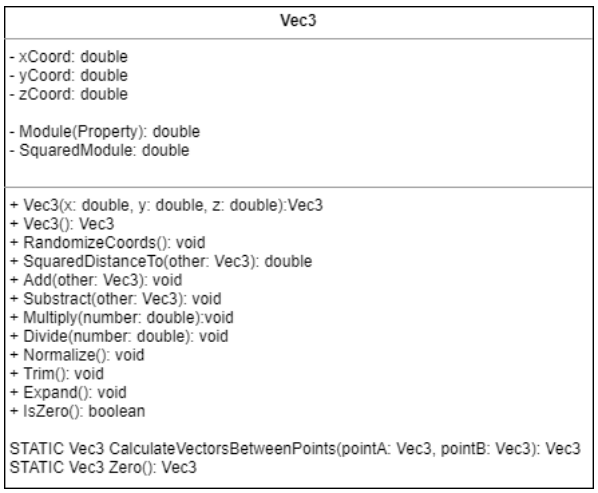

Figura 9.4: Diagrama UML de Vec3 en la etapa 1

La clase **Vec3** modela un vector de $\mathbb{R}^3$ que en este caso se utiliza para guardar las posiciones y velocidades de los animales. Sus atributos son *xCoord, yCoord* y *zCoord* que representan las coordenadas del vector. Se han añadido dos propiedades *Module* y *SquaredModule* que calculan el módulo y el módulo al cuadrado del vector. *SquaredModule* es necesario porque se calculan constantemente distancias entre puntos y comparando las distancias al cuadrado la simulación se ahorra la operación de raíz cuadrada lo que mejora su rendimiento.

Las operaciones básicas que se pueden realizar con los vectores se implementan en los métodos de *Add(), Substract(), Multiply()* y *Divide()*. También se han implementados los métodos *Normalize(), Trim()* y *Expand()* para normalizar, acortar o expandir un vector

respectivamente. El método estático de *CalculateVectorBetweenPoints()* nos devuelve un vector que une dos puntos dados y *Zero()* nos devuelve el vector $\vec{0}$.

9.3.2 Vista

Primeramente se detalla la clase **Vista** en la figura 9.5.

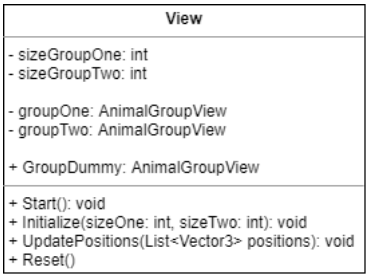

Figura 9.5: Diagrama UML de View en la etapa 1

Esta clase contiene la vista de los dos grupos de animales y guarda sus tamaños. La propiedad *GroupDummy* es necesaria por la manera de instanciar *Prefabs* con la que funciona Unity. Esta clase recibe las posiciones de los animales del controlador y la actualiza llamando al método *UpdatePositions()*. El método *Reset* se utiliza para resetear la vista en cada iteración de la simulación.

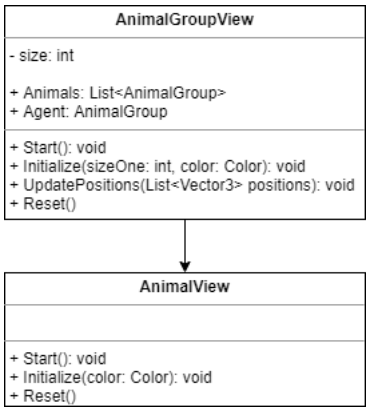

Figura 9.6: Diagrama UML de AnimalGroupView y AnimalView en la etapa 1

En la figura 9.6 se muestran las clases **AnimalGroupView** y **AnimalView**. **AnimalGroupView** tiene una lista de las instancias de **AnimalView** a las que comunica sus nuevas posiciones cada frame.

9.3.3 Controlador

Por último sólo queda mostrar la clase controlador. Como se ve en el diagrama UML,

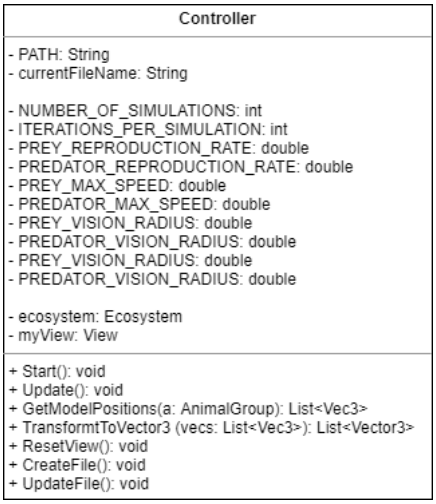

Figura 9.7: Diagrama UML de Controller en etapa 1

contiene todos los parámetros de la simulación además de una ruta para guardar los datos de las simulaciones en ficheros. En cada frame, se llama al método *Update()* que a su vez llama a *Update()* de la clase **Ecosystem** que actualiza las posiciones de los animales en el modelo. Luego obtiene esas posiciones con *GetModelPositions()* y las transforma a instancias de la clase **Vector3**, los vectores de Unity, con el método *TransformToVector3()* y se las comunica a la vista.

9.4 Principios SOLID

En esta sección se analizará el cumplimiento de los principios SOLID.

9.4.1 Principio de Responsabilidad Única

Al haber diseñado el sistema con los principios SOLID en mente se ha aplicado el patrón MVC en la arquitectura de la aplicación para cumplir con este principio. Se han conseguido separar en partes bien diferenciadas las funcionalidades del sistema. El modelo realizará los cálculos, mientras que el controlador podrá definir los parámetros iniciales de cada simulación además de hacer de puente con la vista que representará visualmente

cada experimento.

A un nivel más bajo, también se cumple este principio gracias al uso de patrones de diseño y herencia. La implementación del patrón State en la clase **Animal** es un claro ejemplo de ello porque se separan los posibles comportamientos de un animal en clases distintas.

Existe una excepción con la clase **AnimalFleeState**: como se ha detallado en en la memoria del TFG de Matemáticas en la sección 6.1, esta clase se encarga de calcular el vector fuerza que actúa sobre la presa en cada instante, para calcularlo utiliza una serie de métodos modelan cada una de las reglas de Boids. Todos estos métodos son muy similares entre sí, dado que reciben como parámetros una lista de instancias de la clase **Animal** devuelven una instancia de la clase **Vec3**. Podría existir una clase para cada una de las reglas y unirlas en una superclase utilizando el patrón Composite. De esta manera la clase **AnimalFleeState** podría activar, desactivar e incluso añadir reglas si fuese necesario además de dividir en partes más pequeñas la responsabilidad de dicha clase.

9.4.2 Principio abierto/cerrado

Este principio se cumple ya que se pueden añadir más fases a la simulación o nuevos estados animales sin necesidad de eliminar o refactorizar el código existente. El resultado de haber empleado este principio en el diseño es la utilización del patrón *State*.
Sin embargo, el diseño actual es un prototipo y sufrirá cambios drásticos en futuras etapas de desarrollo. Si este fuera el diseño final sí que se cumpliría el principio.

9.4.3 Principio de Sustitución de Liskov

En este momento las únicas herencias que existen son las que provienen de clases abstractas (**SimulationState** y **AnimalState**), y todas sus clases hijas deben implementar una serie de métodos abstractos obligatoriamente, por lo tanto, este principio se cumple.

9.4.4 Principio de Segregación de Interfaces

Aunque en esta fase no hay interfaces como tal, se podría ver la herencia como una forma de interfaz y habiendo tenido este principio en cuenta desde un principio, se han conseguido segregar las distintas funcionalidades comunes que existen entre los componentes del sistema.

9.4.5 Principio de Inversión de Dependecias

Este principio se cumple, ya que se han implementado clases abstractas y el funcionamiento del sistema no depende de clases concretas.

9.5 Resultados

En esta sección se mostrarán los resultados de dos experimentos distintos.

9.5.1 Experimento 1: Extinción

En primer lugar se realiza un análisis de un experimento con las siguientes condiciones iniciales:

	REPRODUCTION_RATE	VISION_RADIUS	MAX_SPEED	GROUP_SIZE
PREY	1	25	0'58	50
PREDATOR	0'6	15	0'6	15

Tabla 9.1: Parámetros Experimento 1, Etapa 1

Y se obtiene el siguiente resultado:

Se observa que inicialmente la mayoría de los predadores consiguen cazar aunque al tener una probabilidad de reproducción baja acaban desapareciendo. En cuanto a las presas se ve que al comienzo su población disminuye hasta casi la mitad de la inicial pero gracias al descenso de número de predadores acaba aumentando exponencialmente. La extinción de los predadores puede haberse producido, entre otros motivos, por la gran

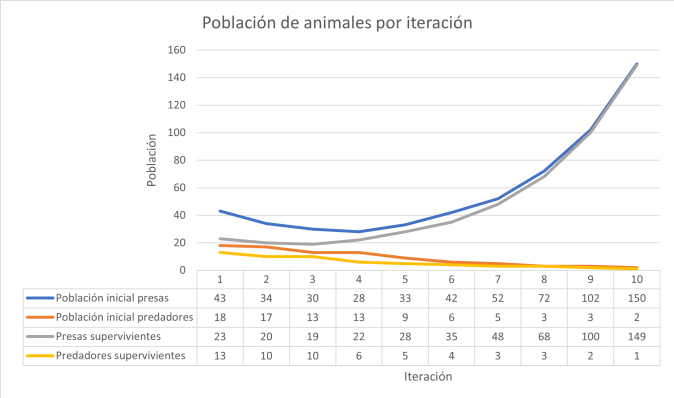

Figura 9.8: Gráfico de poblaciones del experimento 1

diferencia con el radio de visión de las presas. Al tener éstas un radio mucho mayor, pueden reaccionar antes a los movimientos de los predadores y hacer que la balanza se decante a su favor pese a tener una velocidad máxima menor.

9.5.2 Experimento 2: Equilibrio

Tras probar con distintas condiciones iniciales, se consigue llegar a un sistema que alcanza el equilibrio.

Eligiendo los siguientes valores:

	REPRODUCTION_RATE	VISION_RADIUS	MAX_SPEED	GROUP_SIZE
PREY	1	8	0'5	500
PREDATOR	1	15	0'6	15

Tabla 9.2: Parámetros Experimento 2, Etapa 1

Se muestra el resultado de la simulación en la figura 9.9, donde se puede observar que la población de presas disminuye drásticamente porque los predadores no compiten entre ellos por el alimento al ser un grupo de tamaño reducido. En la iteración 11, la población de predadores alcanza un pico dado que la competición entre ellos es más fuerte al haber disminuido el número de presas disponibles. Es a partir de esta iteración cuando se llega a un equilibrio, ambas especies se estabilizan y crecen a un ritmo constante. No obstante,

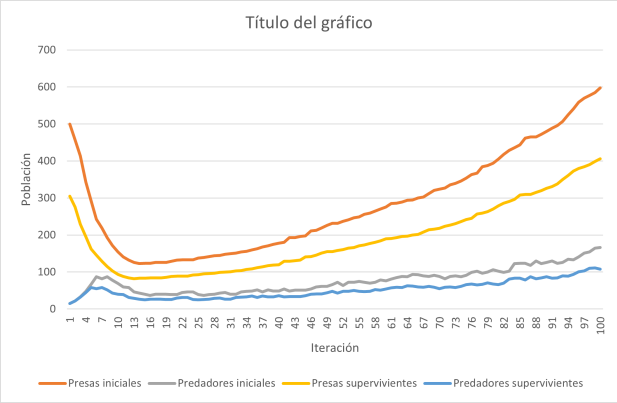

Figura 9.9: Gráfico de poblaciones del experimento 2

se aprecia que el ritmo de crecimiento de las presas es mayor que el de los predadores. Esto puede deberse a que las presas no compiten entre ellas al no necesitar alimento para sobrevivir, simplemente llegar a una zona segura.

9.6 Refactorizaciones

A partir del análisis del estado del sistema realizado en las secciones 9.4 y 9.5 se decide realizar los siguientes cambios en el código:

- Aplicar el patrón Composite en la clase **AnimalFleeState** para cumplir el Principio de Responsabilidad Única.
- Añadir a la clase **Ecosytem** un atributo de la clase **Grass**, recurso por el que competirán las presas, para conseguir una dinámica poblacional más realista.

Capítulo 10

Etapa 2: Implementación de estrategias de caza

Esta tercera etapa ya es la etapa final. Después de los experimentos y análisis realizados en la etapa 1, se comienza refinando el diseño del sistema para más tarde modelar las distintas estrategias de caza de los depredadores.
En primer lugar, se definirán los objetivos de la etapa. A continuación se detallará el diseño con ayuda de modelos UML a la vez que se explican los patrones utilizados y las diferencias en cuanto a la etapa 1. Posteriormente se analizará el cumplimiento de los principios SOLID. Para concluir el capítulo se realizarán experimentos en la simulación para definir los objetivos de la siguiente etapa y detectar refactorizaciones necesarias para futuras etapas.

10.1 Objetivos

En esta etapa se fijan objetivos más pequeños con propósito de disminuir la cantidad de trabajo en con respecto a la anterior.

- Implementar las refactorizaciones especificadas en la sección 9.6.
- Implementar una estrategia de caza con el algoritmo *PSO*.
- Implementar una estrategia de caza con el algoritmo *GWO*.

- Implementar una estrategia de caza con el algoritmo *WOA*.
- Conclusiones de la etapa.

10.2 Estado de la simulación

En esta etapa la simulación es esencialmente la misma que en la etapa 1, excepto por el modo de alimentarse que tienen los animales. Ahora los animales comparten recursos a los que acceden para comer, lo hacen antes de reproducirse.

1. Inicialización de los parámetros iniciales de la simulación, que son:
   - **ITERATIONS**: Número de iteraciones de la simulación.
   - **PREY_PARAMETERS**: Contienen la probabilidad de reproducción, máxima velocidad, radio de visión y tamaño del grupo de las presas.
   - **PREY_PARAMETERS**: Contienen la probabilidad de reproducción, máxima velocidad, radio de visión y tamaño del grupo de los depredadores.
   - **INITIAL_PLANTS**: El número inicial de plantas, el recurso de las presas.
   - **GROWTH_RATE**: El ratio de creciemiento de las plantas.
   - **THRESHOLD**: Un límite inferior de la cantidad de plantas que puede haber.
2. Se inicializa el ecosistema en el estado **SimulationSurviveState**
3. Se inicializa un **AnimalGroup** cuyas instancias de **Animal** tienen los atributos definidos en **PREY_PARAMETERS** y con un estado inicial **AnimalFleeState**. Este grupo representa a las presas.
4. Se inicializa un **AnimalGroup** cuyas instancias de **Animal** tienen los atributos definidos en **PREDATOR_PARAMETERS** y con un estado inicial **AnimalHuntState**. Este grupo representa a los depredadores.
5. Las presas huyen hacia una zona segura mientras que los predadores intentan cazarlas. Si una presa es cazada, se añaden recursos que consumir a la manada de depredadores.

6. Cuando todas las presas que no han sido cazadas han llegado a la zona segura, el ecosistema pasa al estado **SimulationEvolveState**. En este estado los animales se reproducen de la siguiente manera:

    - Todas las presas que hayan alcanzado la zona segura intentan alimentarse con los recursos almacenados en su grupo, las que consigan hacerlo sobreviven para reproducirse. Por cada pareja existente, se intenta crear otra presa según la probabilidad de reproducción definida en **PREY_PARAMETERS**.
    - Los depredadores intentan alimentarse con los recursos acumulados en la manada, las que consigan hacerlo sobreviven para reproducirse. Por cada pareja existente, se intenta crear otra presa según la probabilidad de reproducción definida en **PREDATOR_PARAMETERS**.

7. Se finaliza la iteración y se vuelve al paso 3 aunque los grupos tendrán un tamaño inicial igual a los supervivientes + nacidos en esta iteración. Si el número de iteraciones que ha hecho la simulación es igual a el parámetro **ITERATIONS**, entonces finaliza.

La simulación puede terminar antes de realizar el número indicado de iteraciones si uno de los dos grupos de animales se extingue.

Además de todo esto, asumiremos que 1 segundo equivalen a 30 frames en la simulación.

10.3 Diseño

En esta sección se analizará la evolución que ha tenido la arquitectura del sistema tras haber cumplido los objetivos de la etapa 2. El diagrama UML completo se puede consultar en el apéndice A.3.

10.3.1 Modelo

Como se muestra en la figura 10.1, la clase **Ecosystem** no ha sufrido cambios ya que las dos principales fases de la simulación siguen siendo las mismas, modeladas por las clases **SimulationEvolveState** y **SimulationSurviveState**.

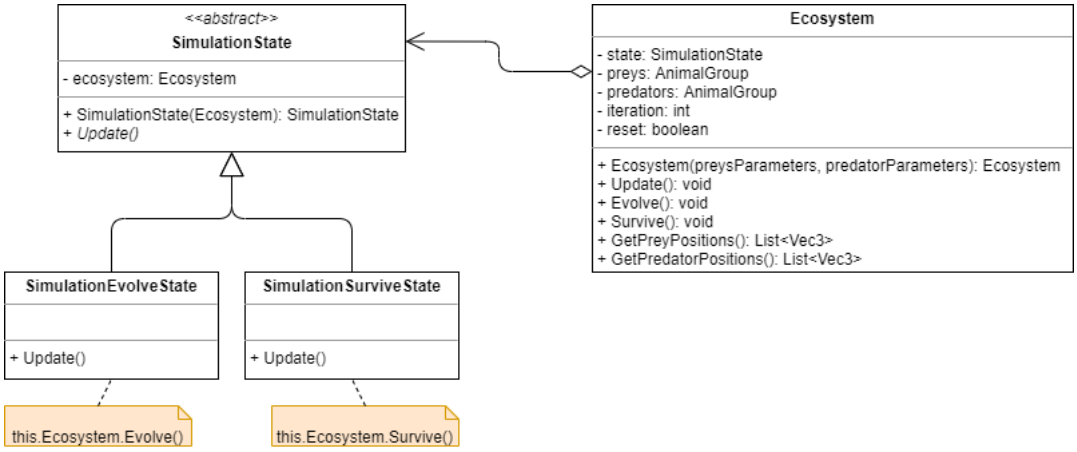

Figura 10.1: Diagrama UML de Ecosystem en la etapa 2

Una de las clases que más ha evolucinado en esta iteración ha sido **AnimalGroup** ya que ha adquirido varios componentes expuestos en la figura 10.2. Ahora **AnimalGroup**

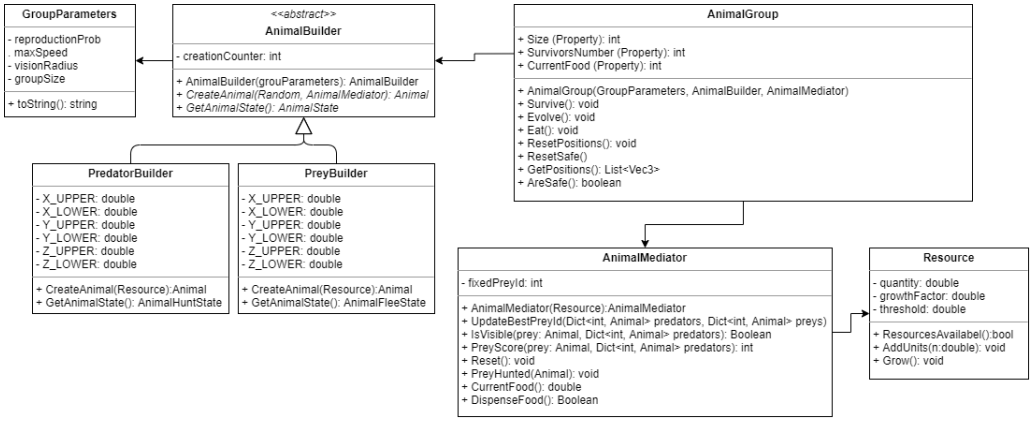

Figura 10.2: Diagrama UML de AnimalGroup en la etapa 2

está compuesto por:

1. **AnimalMediator**: Dado que en esta etapa los animales de un mismo grupo intercambian información, como por ejemplo a qué presa cazar, ha sido necesario crear una forma de comunicación entre ellos. Esta clase también se encarga de almacenar y dispensar alimentos a la clase **Animal**. Detalles de su implementación:

   - *fixedPreyId* es el id de la presa a la que cazarán los depredadores
   - *UpdateBestPrey()* actualiza el id de la presa seleccionando a la que sea visible por más depredadores utilizando las funciones *IsVisible()* y *PreyScore()*.

- *Reset()* resetea el id de la presa seleccionada y hacer crecer los recursos.
- *PreyHunted()* añade cantidad a **Resource** según la presa cazada. Cuando los depredadores usan una estrategia en grupo se añaden 3 unidades de recursos pero cuando cazan solos no se añade ninguna y simplemente se marca al depredador como superviviente con la variable *hasEaten*.
- *CurrentFood()* devuelve la cantidad actual de recursos.
- *DispenseFood()* devuelve true si quedan recursos y resta 1 unidad de los mismos.

2. **Resource**: Esta es la clase que modela los recursos con los que los animales se alimentarán. En el caso de las presas representa los vegetales y en el de los depredadores las presas cazadas (su ratio de crecimiento será pues 0). Esta clase responde a la segunda refactorización propuesta en la sección 9.6. En vez de añadir una clase **Grass** a Ecosystem se ha optado por implementar un objeto que sirva para toda instancia **AnimalGroup** sin importar el grupo que representen. Sus métodos son:

   - *ResourcesAvailable()* que resta 1 unidad a *quantity* y devuelve true si no es menor que *threshold*.
   - *AddUnits()* suma la entrada a *quantity*.
   - *Grow()* multiplica *quantity* por *growthRate* al final de la fase de evolución.

3. **AnimalBuilder**: Clase que se encarga de crear instancias del tipo **Animal** con el método *CreateAnimal()* y que devuelve el estado inicial del mismo con *GetAnimalState()*. Al modelarse la distinción de comportamiento con estados en vez de con clases que hereden de **Animal** son necesarias dos tipos de constructores: **PreyBuilder** que inicialize a los animales con el estado **AnimalFleeState** y **PredatorBuilder** que inicialize a los animales con el estado **AnimalHuntState**.

4. **GroupParameters**: Es la clase que agrupa los parámetros que antes se encontraban en **AnimalGroup** que son: *reproductionProb*, *maxSpeed*, *visionRadius*, *groupSize*

Es necesario recalcar que el nombre de **AnimalBuilder** puede confundir en cuanto al patrón que utiliza, no es el Builder sino el Factory.

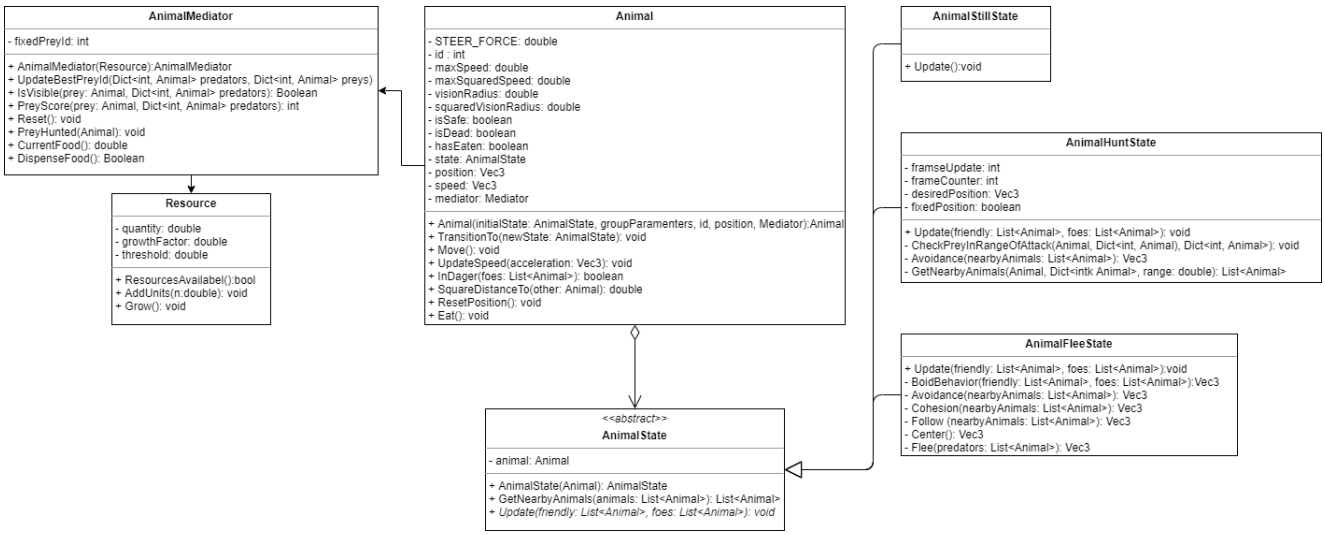

Figura 10.3: Diagrama UML de Animal en la etapa 2

Se prosigue explicando la transformación de la clase **Animal** ilustrada en la figura 10.3:

1. Ahora cada **Animal** conoce al **AnimalMediator** del grupo al que pertenece.

2. Se ha implementado la función *Eat()* para que el animal se alimente. También hay un nuevo atributo relacionado con esta funcionalidad: *hasEaten* que es true si el animal ha consumido recursos en una iteración.

Los estados **AnimalFleeState** y **AnimalHuntState** han sido modificadas drásticamente como se puede observar en las figuras 10.4 y 10.5.

Para cumplir con los objetivos de la sección 9.6 se ha implementado el patrón Composite en la clase **AnimalFleeState** para granular las reglas del algoritmo Boids [1] que simula la huida de las presas:

1. **Boid**: Contiene los pesos para las distintas reglas Boid que puede utilizar y combina sus resultados con el método *CalculateForces()*. Hace las veces de Componente del patrón Composite

[1] Algoritmo detallado en la memoria del TFG de Matemáticas, apartado 6.1

2. **BoidRule**: Clase abstracta que hace las veces de hoja dentro del patrón Composite

3. **AvoidanceRule**: Regla de esquive.

4. **CenterRule**: Regla de centrarse.

5. **FollowRule**: Regla de centrarse.

6. **CohesionRule**: Regla de centrarse.

7. **FleeRule**: Regla de huir.

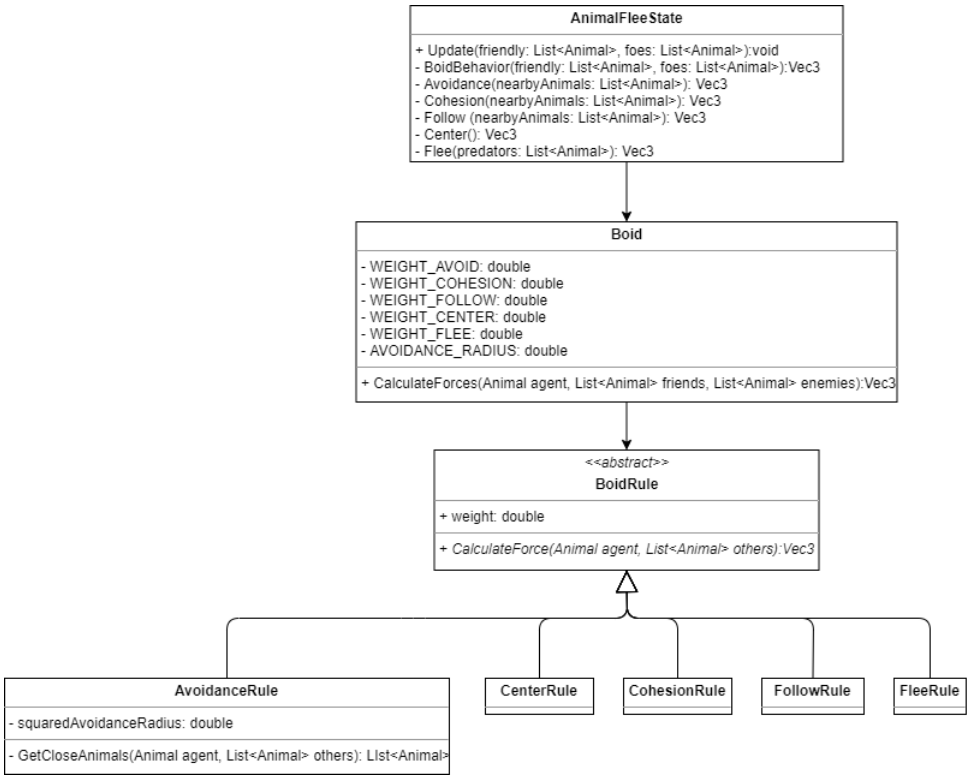

Figura 10.4: Diagrama UML de AnimalFleeState en la etapa 2

Por otro lado, se ha implementado el patrón Strategy en **AnimalHuntState**, ahora esta clase tiene un atributo del tipo **HuntingStrategy** que calcula la posición óptima de un depredador en un momento de la fase de supervivencia dado.

1. **HuntingStrategy**: Clase abstracta que define los métodos y atributos que utilizará **AnimalHuntState** para mover al depredador que tiene asignado.

- *framesUpdate* es el intervalo de *frames* que tarda en actualizarse la posición deseada de un depredador.
- *GetDesiredPosition()* es la función principal de la clase, devuelve una posición a la que el depredador deberá moverse.
- *HasFixedPrey()* es la función que comprueba si el depredador tiene una presa fijada.
- *SelectPrey()* actualiza el id de la presa a cazar.
- *GetFixedPrey()* obtiene el id de la presa a cazar por el depredador.
- *HuntPrey()* comprueba si el depredador está en rango de ataque para cazar a su presa y actualiza los recursos en caso verdadero.

2. **SimpleStrategy**: En esta clase se ha refactorizado el código que se encontraba en **AnimalHuntState** en la etapa anterior para que siga funcionando como una estrategia.

3. **PSOStrategy**: Implementa el algoritmo metaheurístico *Particle Swarm Optimization* [2]. Sus atributos y métodos son necesarios para el mismo.

4. **GWOStrategy**: Implementa el algoritmo metaheurístico *Grey Wolf Optimizer* [3]. Sus atributos y métodos son necesarios para el mismo.

5. **WOAStrategy**: Implementa el algoritmo metaheurístico *Whale Optimizer Algorithm* [4]. Sus atributos y métodos son necesarios para el mismo.

6. **Metaheuristic**: Interfaz que utilizan las estrategias basadas en metaheurísticas. Define los métodos *ObjectiveFunction()* y *CalculateFitness()* que se usan para medir la calidad de las soluciones obtenidas con las distintas estrategias.

[2]La implementación del PSO se detalla en la memoria del TFG de Matemáticas con título "*Simulación de estrategias de caza de animales gregarios a partir de metaheurísticas*" en la sección 6.2

[3]La implementación del GWO se detalla en la memoria del TFG de Matemáticas con título "*Simulación de estrategias de caza de animales gregarios a partir de metaheurísticas*" en la sección 6.3

[4]La implementación del WOA se detalla en la memoria del TFG de Matemáticas con título "*Simulación de estrategias de caza de animales gregarios a partir de metaheurísticas*" en la sección 6.4

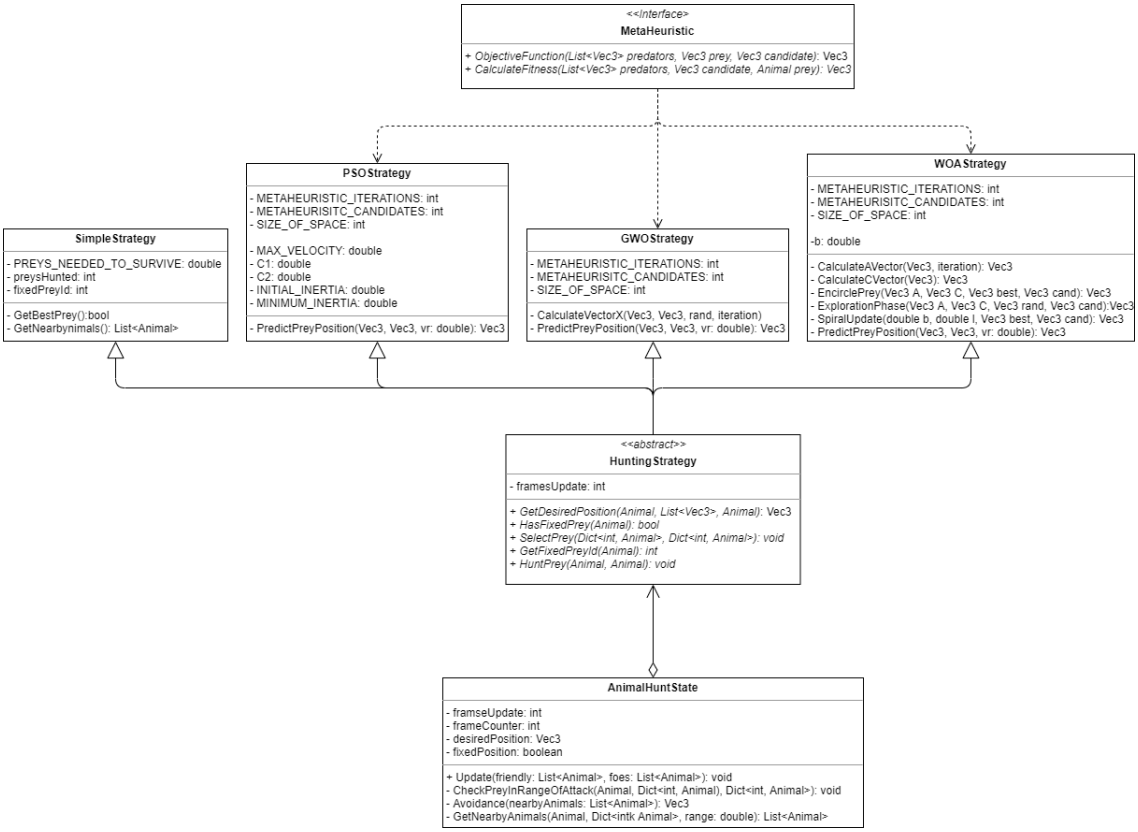

Figura 10.5: Diagrama UML de AnimalHuntState en la etapa 2

Finalmente se muestra la versión de esta etapa de la clase **Vec3** en la figura 10.6, se han añadido los siguiente elementos:

1. *Coordenadas límite* para limitar los valores de las coordenadas al randomizarlas.

2. *hasBounds* para saber si tiene límites o no.

3. *Add()* es un método estático que suma dos vectores.

4. *Substract()* es un método estático que resta dos vectores.

5. *WolfProduct()* es un método estático que modela una operación del algoritmo GWO.

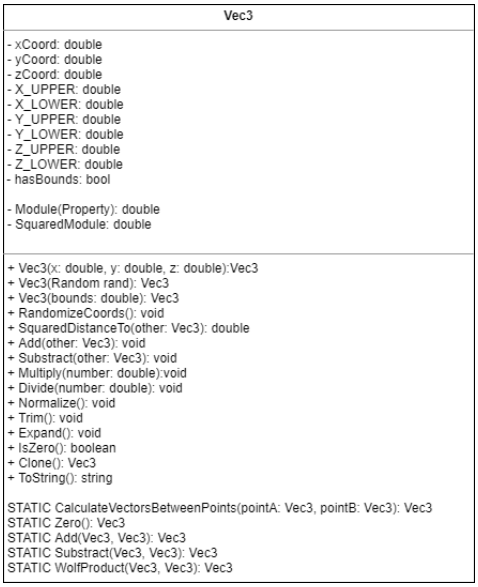

Figura 10.6: Diagrama UML de Vec3 en la etapa 2

10.3.2 Vista

La vista ha mantenido su arquitectura en esta etapa.

10.3.3 Controlador

Ahora Controller tiene atributos del tipo **GroupParameters** y nuevas constantes para las plantas como se ve en la figura 10.7 aunque su funcionamiento sigue siendo el mismo.

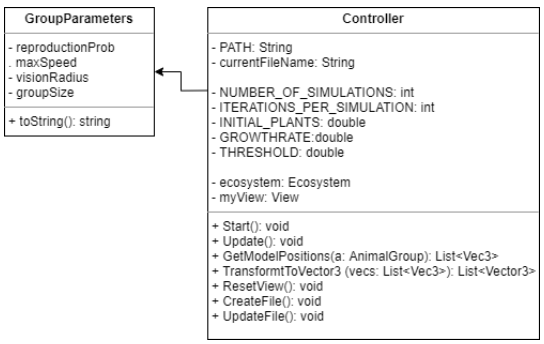

Figura 10.7: Diagrama UML de Controller en la etapa 2

10.4 Principios SOLID

10.4.1 Principio de Responsabilidad Única

Al haber evolucionado el sistema con este principio como guía se ha conseguido mantener la granularidad y cohesión del código. Un ejemplo de esto es la implementación del patrón Composite en la clase **AnimalHuntState**.

10.4.2 Principio de abierto/cerrado

En el estado actual del sistema, se podrían añadir nuevas estrategias de caza, estados de simulación o estados animales sin tener que cambiar nada de lo que ya existe por lo que este principio se sigue cumpliendo en la etapa 2.

10.4.3 Principio de Sustitución de Liskov

De nuevo, implementando el código en base a los principios SOLID, se ha conseguido un buen polimorfismo como por ejemplo en las distintas estrategias implementadas que heredan de la clase **HuntingStrategy** y que usa **AnimalHuntState** sin importar el subtipo utilizado. Se puede decir que este principio también se cumple la etapa actual.

10.4.4 Principio de Segregación de Interfaces

En esta iteración del desarrollo se ha añadido una interfaz : la interfaz **Metaheuristic**, que ha permitido desacoplar las clases **PSOStrategy**, **GWOStrategy** y **WOAStrategy** de la clase **SimpleStrategy**. De no haberse hecho así, SimpleStrategy tendría métodos relacionados con las metaheurísticas que no utilizaría y viceversa. El principio también se cumple en este punto.

10.4.5 Principio de Inversión de Dependencias

La encapsulación y abstracción está presente en todo el código, es por eso que las clases que modelan la simulación pueden usar clases núcleo del sistema de manera clara y eficiente, haciendo así que se cumpla este principio.

10.5 Resultados

A continuación se muestran los experimentos realizados en la etapa 2.

10.5.1 Caza en grupo

Uno de los objetivos de esta iteración del desarrollo era simular las estrategias de caza en grupo. El resultado no ha sido demasiado satisfactorio, ya que los depredadores acaban extinguiéndose en la mayoría de experimentos. Esto es debido a que en algunas iteraciones no son capaces de atrapar a ninguna presa por lo que no consiguen recursos para alimentarse y se extinguen.

Uno de los experimentos más duraderos se ha conseguido con las siguiente configuración:

	REPRODUCTION_RATE	VISION_RADIUS	MAX_SPEED	GROUP_SIZE
PREY	1	10	0'38	20
PREDATOR	1	50	0'45	7

Tabla 10.1: Parámetros Animales Experimento 1, Etapa 2

INITIAL_PLANTS	GROWTH_RATE	THRESHOLD
55	1'5	10

Tabla 10.2: Parámetros Plantas Experimento 1, Etapa 2.

El resultado del experimento 1 se puede ver en la figura 10.8

Como se ve el número de depredadores se mantiene constante, consiguen cazar lo justo para sobrevivir por lo que las presas comienzan con un crecimiento exponencial que se ve limitado por las plantas que hay. Cuando la vegetación comienza a bajar hasta su mínimo, también lo hace la población de presas durante un tiempo. Al final las plantas vuelven a crecer exponencialmente, justo al mismo tiempo en el que se acaba la simulación porque los depredadores no cazan exitosamente y acaban por extinguirse.

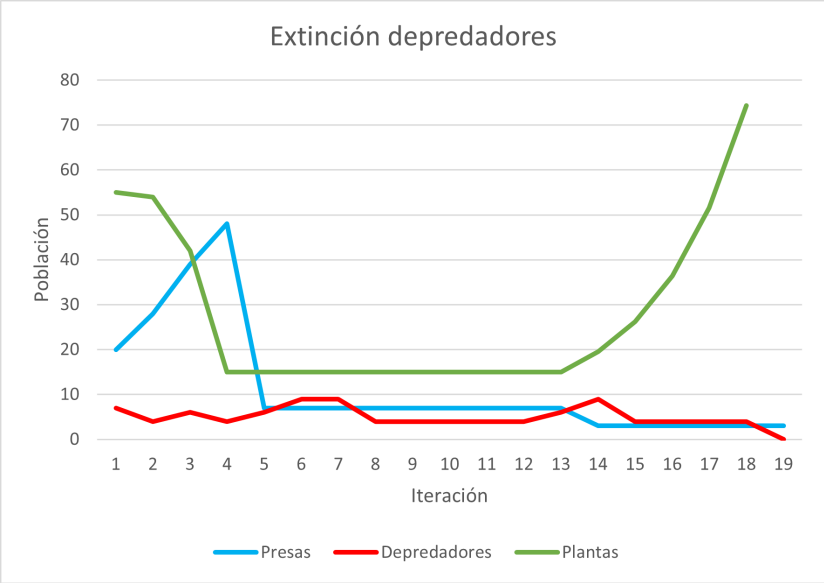

Figura 10.8: Gráfico de poblaciones del experimento 1

Otro experimento que ha sido más duradero que los demás ha sido el siguiente en el que se extinguen las presas:

	REPRODUCTION_RATE	VISION_RADIUS	MAX_SPEED	GROUP_SIZE
PREY	1	10	0'38	20
PREDATOR	1	50	0'47	7

Tabla 10.3: Parámetros Animales Experimento 2, Etapa 2

INITIAL_PLANTS	GROWTH_RATE	THRESHOLD
40	1'5	10

Tabla 10.4: Parámetros Plantas Experimento 2, Etapa 2.

El resultado del experimento 2 se puede ver en la figura 10.9. En el momento en que las presas sobrepasan la población de plantas, paran su crecimiento y entran en equilibrio con los depredadores por unas pocas iteraciones hasta que son superadas en número y los depredadores acaban por cazar a todas. Las plantas crecen libremente una vez la población de presas es inferior a la suya.

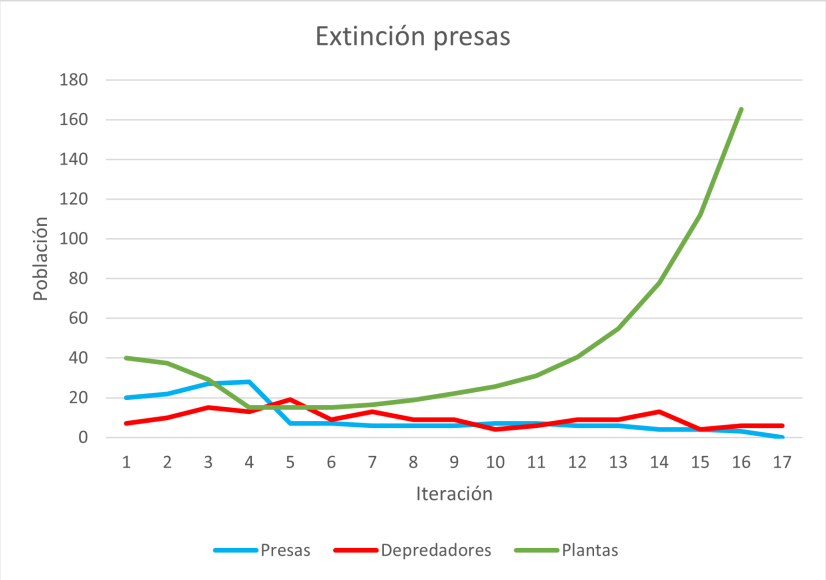

Figura 10.9: Gráfico de poblaciones del experimento 2

Para evitar los problemas de extinción que tienen los depredadores se podría hacer que no necesitasen comer en cada iteración, sino cada dos.

10.5.2 Caza individual

A continuación se muestra qué ocurre si los depredadores utilizan la estrategia simple en la que cada uno va a por una presa distinta, ahora que las presas también necesitan comer para sobrevivir.

Los parámetros del experimento 3 son:

	REPRODUCTION_RATE	VISION_RADIUS	MAX_SPEED	GROUP_SIZE
PREY	0'9	12	0'54	200
PREDATOR	0'2	30	0'6	8

Tabla 10.5: Parámetros Animales Experimento 3, Etapa 2

INITIAL_PLANTS	GROWTH_RATE	THRESHOLD
1100	1'4	500

Tabla 10.6: Parámetros Plantas Experimento 3, Etapa 2.

En la figura 10.10 se pueden observar los resultados.

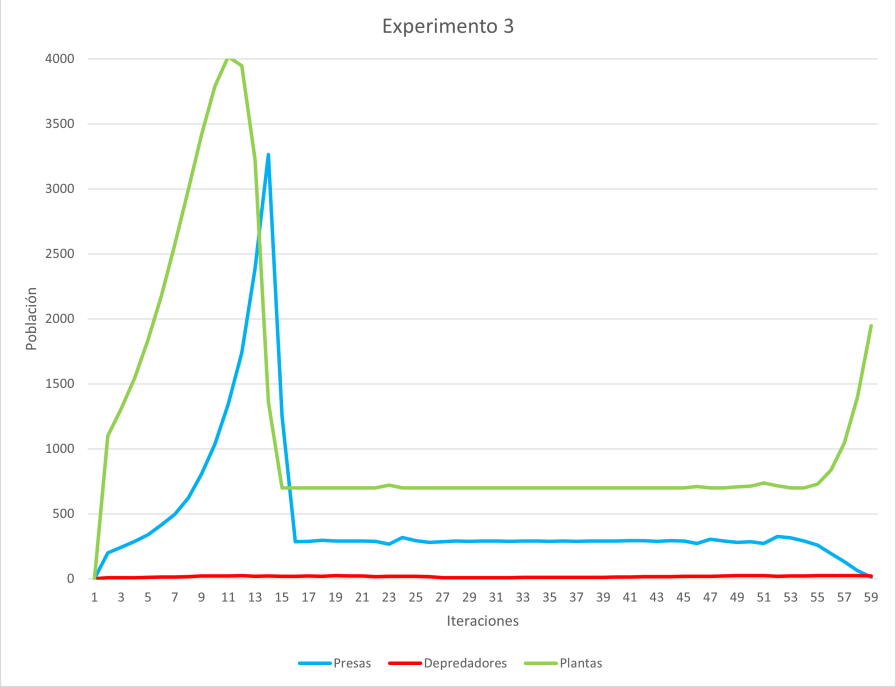

Figura 10.10: Gráfico de poblaciones del experimento 3

La población de depredadores oscila entre los 10 y los 30 individuos a lo largo de todo el experimento ya que no tienen un ratio de reproducción muy elevado, esto hace que la proporción de presas cazada sea más o menos la misma en cada iteración. Aun así las presas superan en número a las plantas por lo que la sobreexplotan y se activa el threshold de las mismas. A partir de este momento las presas no disponen de recursos suficientes para crecer por lo que es cuestión de tiempo que los depredadores tengan suerte y cacen a un número mayor de presas de lo habitual, que es lo que ocurre a partir de la iteración 50. Las presas a acaban por extinguirse y por lo tanto lo harán los depredadores.

10.6 Refactorizaciones

En esta etapa el código es sólido y no se han detectado antipatrones. A partir de este punto de desarrollo es sería fácil evolucionar el sistema añadiendo nuevos estados animales o fases de simulación sin necesidad de alterar el código actual. Se decide que no son necesarias refactorizaciones en esta etapa.

Capítulo 11

Conclusiones y trabajos futuros

En este capítulo se realiza una retrospectiva sobre el proyecto y su evolución y se propone una nueva etapa de desarrollo.

11.1 Conclusiones

Tras plantear la idea original del proyecto, comenzó una etapa de investigación acerca de simulaciones de sistemas predador-presa con la que se obtuvieron las bases para crear una aplicación que permitiese cumplir los objetivos definidos.

Se decidió utilizar el motor gráfico *Unity* con el que fue necesario familiarizarse. Utilizar el motor gráfico ha sido más útil de lo que podría parecer ya que no se habrían detectado bugs sin la representación visual del modelo, por ejemplo, ocurrió que el método que sumaba vectores en la clase *Vec3* se implementó de manera errónea y sumaba la coordenada x de un vector con la coordenada z de otro, lo que hacía que los animales se movieran de una manera extraña, lo que provocó una revisión del código que permitió descubrir el fallo.

Más tarde se estudiaron patrones que podrían ser útiles a la hora de crear el diseño de la aplicación. Gracias a usar un enfoque iterativo se ha podido ir modificando la estructura definida en la etapa 0 una vez detectadas mejores formas de implementar ciertas funcionalidades. La arquitectura resultante es distinta a la diseñada incialmente, aun así sigue siendo limpia y eficiente, de hecho, el modelo podría funcionar sin necesidad de *Unity*, lo que es indicador de un buen diseño.

El desarrollo en espiral también ha permitido reconducir el comportamiento de la simulación hacia el inicialmente planteado, aunque el funcionamiento final de la misma no sea exactamente igual al definido en el capítulo 5. Era un sistema demasiado complejo y el resultado final ha surgido de ir iterando según las conclusiones obtenidas de los experimentos realizados. Quizás con más tiempo se hubiera llegado al sistema deseado. Aún así, la aplicación desarrollada es lo suficientemente compleja para poder observar propiedades emergentes en el estudio de las dinámicas poblacionales e incluso en la simulación de caza en tiempo real. Encontrar configuraciones idóneas de parámetros para encontrar puntos de equilibrio en la simulación ha sido una tarea complicada porque el sistema tiene demasiados parámetros por lo que hay que ir modificándolos de manera gradual hasta dar con la clave para conseguir una simulación que se alargue en el tiempo. Pese a todo, es satisfactorio dar con esta combinaciones ya que se pueden ver las dinámicas de población que se dan en la naturaleza.
En cuanto a rendimiento, el sistema necesita una gran capacidad computacional para realizar los cálculos, hacer experimentos con más de 500 animales ralentiza considerablemente la simulación por lo que puede que no sea la manera idónea estudiar un sistema-predador presa con un número elevado de agentes. Una opción para resolver esto sería usar los patrones de optimización de los que se habla previamente.

11.2 Trabajos futuros

Se propone implementar la funcionalidad de reproducción definida en el apartado 5.3.2. Ya que el funcionamiento del sistema es ligeramente distinto al propuesto incialmente se reformula la funcionalidad de la siguiente manera:

- Cada animal tiene un pseudo-ADN en forma de una cadena de 10 caracteres con distintas letras que definirán las características del animal. Por ejemplo: VRVRRRRRRVV (V para velocidad, R radio de visión) significará que el individuo tiene velocidad 4 y radio de visión 6. Los parámetros de **GroupAnimal** serán la base de la que parten los animales y los caracteres de este ADN se potenciarán los distintas características que poseen.

- Un adulto tiene todos los caracteres activos.

- Una cría sólo tiene los 5 primeros caracteres activos.
- Si una cría sobrevive una iteración pasa a ser adulto
- Dos adultos supervivientes generan una cría con el ADN resultante de un algoritmo genético que combina los ADN de sus progenitores.

Una idea para el diseño es la que se ve en la figura 11.1.

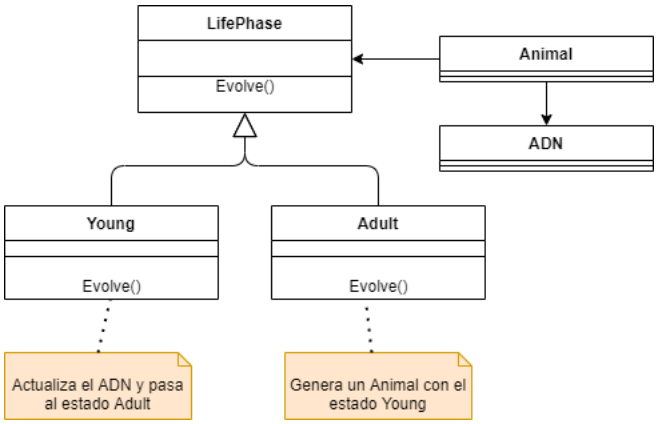

Figura 11.1: Posible diseño para la funcionalidad de reproducción

Se propone usar otro patrón State de tal forma que **Animal** llama al método *Evolve()* implementado en los distintos estados. Además ahora **Animal** tendrá una clase **ADN**.

Para más adelante, tener una interfaz de usuario que presente los datos de la simulación en tiempo real en vez de guardarlos en ficheros externos sería una forma de mejorar el análisis de los experimentos.

Apéndice A

Diagramas UML completos

A.1 Etapa 0

Figura A.1: Diagrama UML del sistema en la etapa 0

A.2 Etapa 1

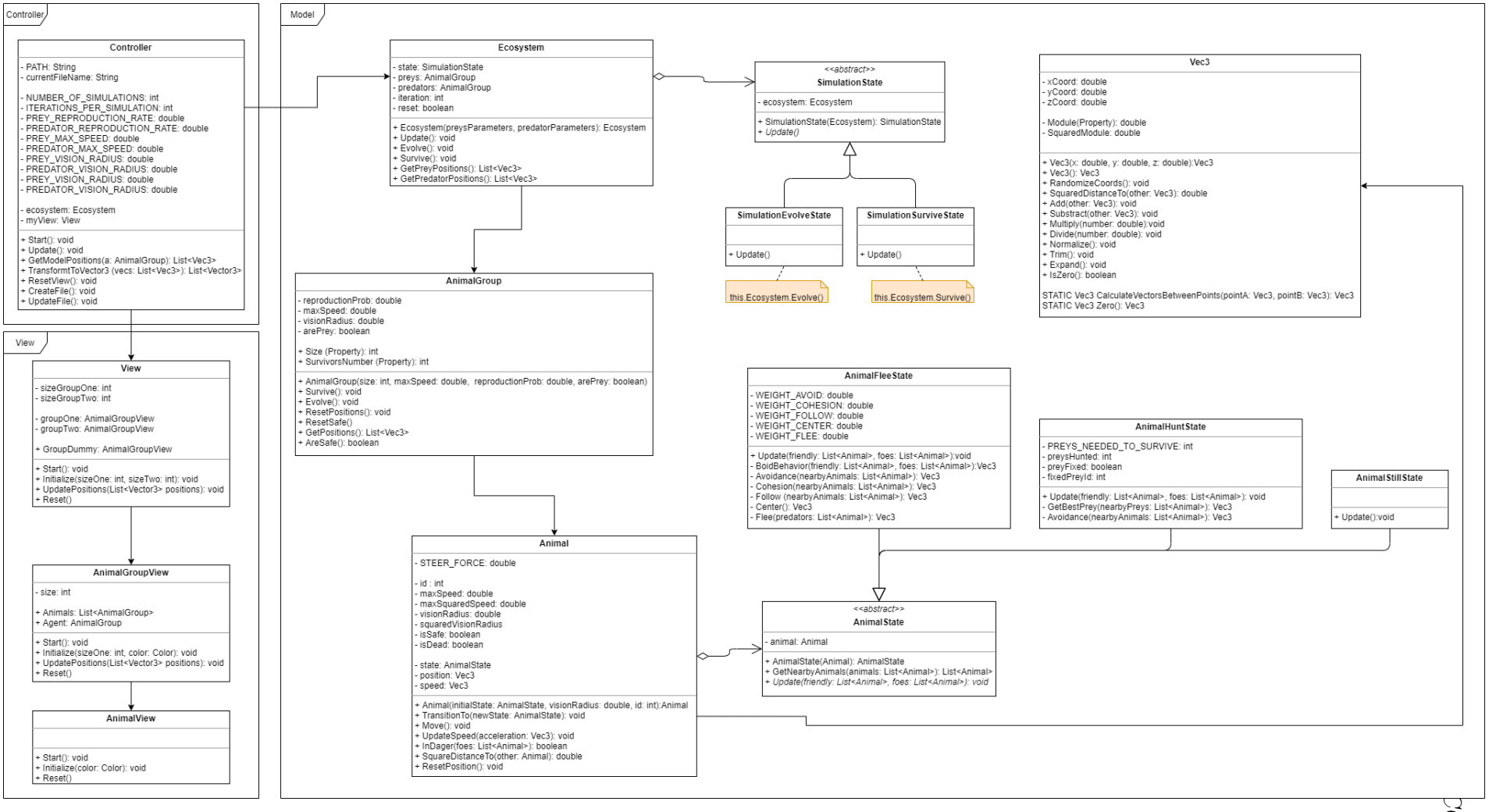

Figura A.2: Diagrama UML del sistema en la etapa 1

A.3 Etapa 2

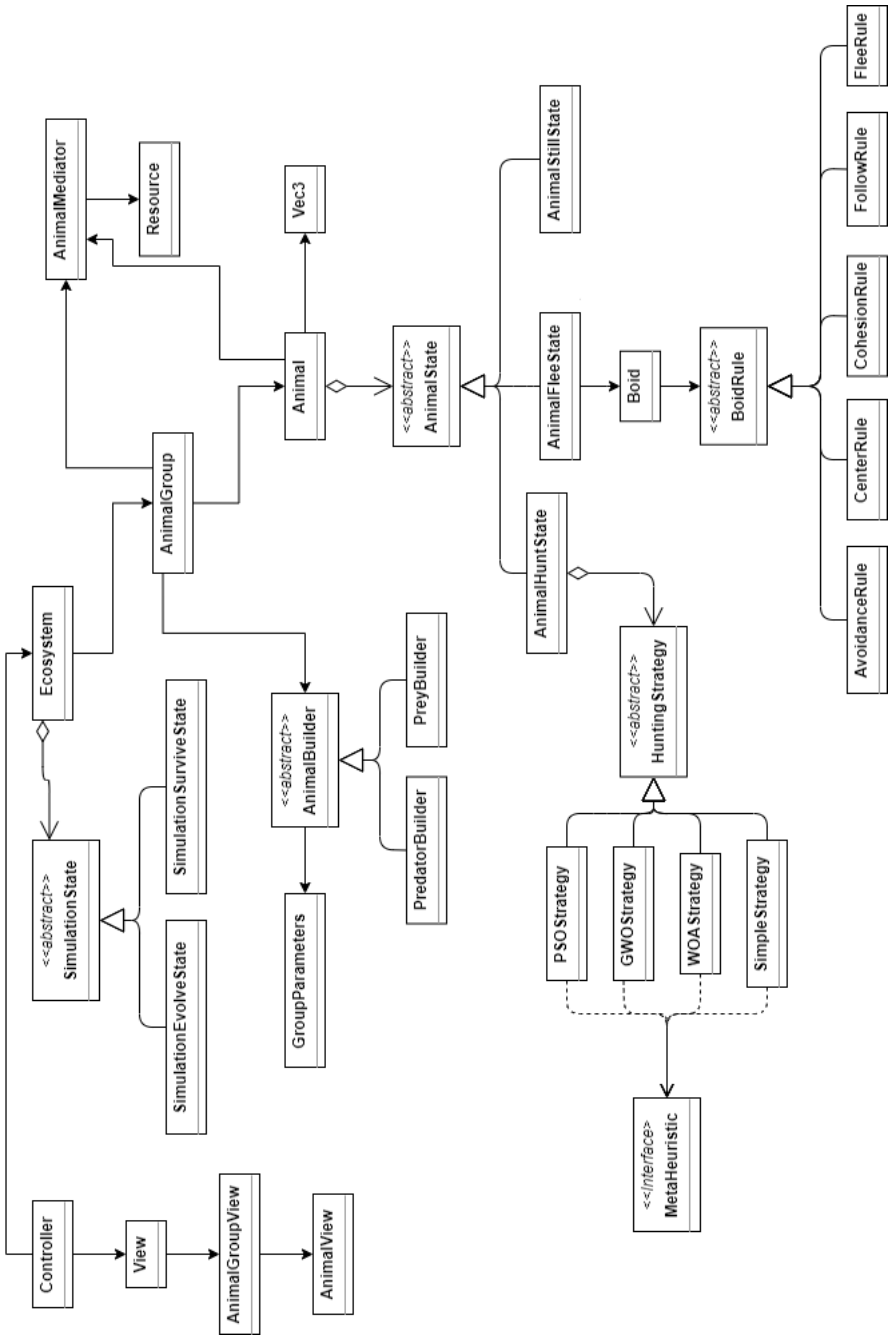

Figura A.3: Diagrama UML simplificado del sistema en la etapa 2

Apéndice B

Manual de usuario

Guía rápida para usar la aplicación.

B.1 Descarga e instalación

Si no dispone del archivo .zip con nombre "*EcoSimPhase2DanielCarmonaPedrajas.zip*" acceda al siguiente enlace y descarge el archivo pinchando sobre él como se muestra en la figura B.1.

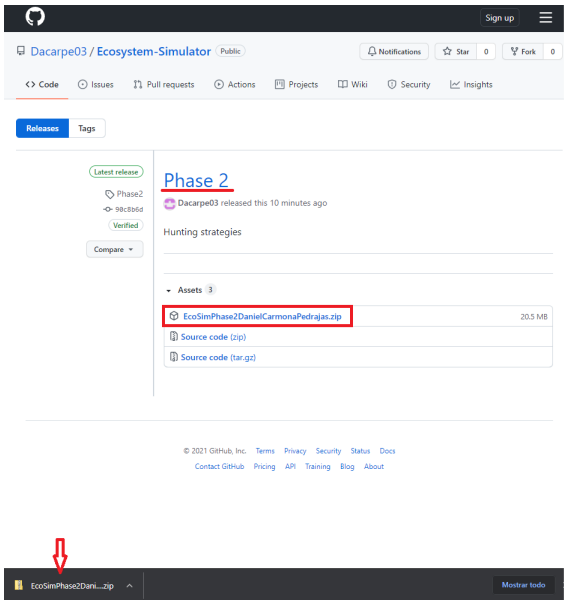

Figura B.1: Guía de descarga

Una vez descargado el archivo es necesario descomprimirlo. A continuación abra la

carpeta y haga doble click en el archivo *EcoSim* (ver figura B.2) para ejecutarlo. Antes de abrirlo, aclarar que la aplicación sólo puede cerrarse utilizando la combinación de teclas *Alt+F4*.

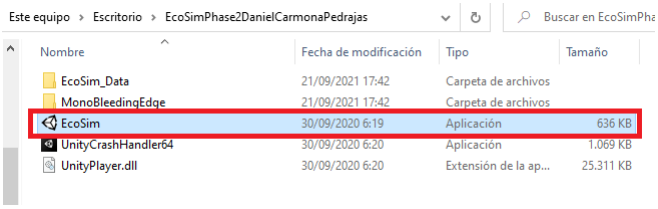

Figura B.2: Abrir el archivo

B.2 Menú de configuración

Al abrir el archivo se abrirá un menú como el que se presenta en la figura B.3.

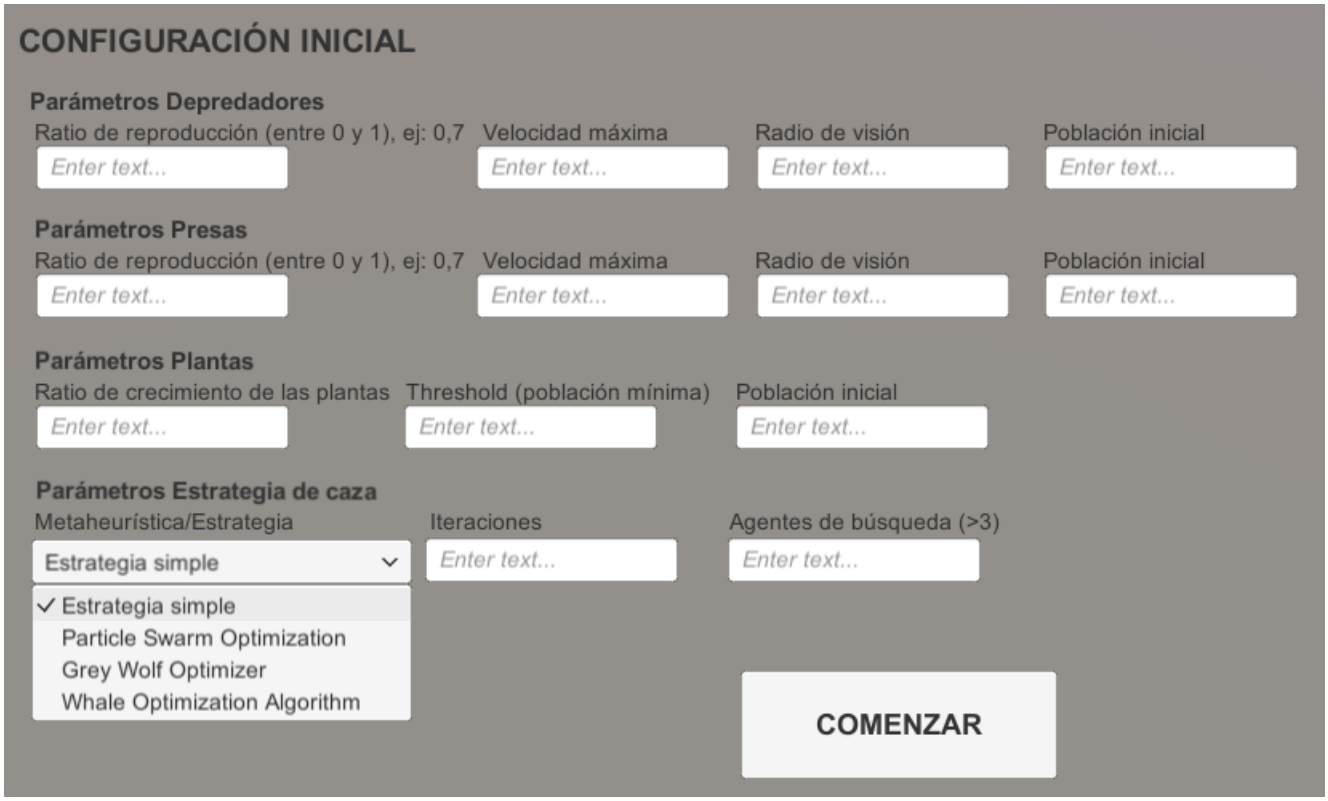

Figura B.3: Menú de configuración

Para cada parámetro relevante en la aplicación se ha añadido la posibilidad de definirlo manualmente por el usuario. Hay un campo de texto que rellenar. Si un campo se deja

en blanco se tomará un valor por defecto por lo que no es necesario rellenar todos los espacios. Es importante que los números decimales se escriban con comas en vez de con puntos, por ejemplo, 0,55.

Los parámetros de depredadores y presas funcionan de la misma forma.

- **Ratio de reproducción** es la probabilidad de que una pareja de supervivientes generen otro animal.
  - Valor por defecto para los **depredadores**: 0,3
  - Valor por defecto para las **presas**: 0,9
- **Velocidad máxima** es para limitar la velocidad que pueden alcanzar los animales. Es necesario tener en cuenta que 30 frames equivalen a 1 segundo, por lo tanto si se quiere hacer que los animales se muevan a 30 unidades por segundo, la velocidad máxima deberá ser 0,1.
  - Valor por defecto para los **depredadores**: 0,6
  - Valor por defecto para las **presas**: 0,55
- **Radio de visión** es el número de unidades de distancia a las que un animal puede detectar a otro, ya pertenezca o no a su misma especie.
  - Valor por defecto para los **depredadores**: 30
  - Valor por defecto para las **presas**: 8
- **Población inicial** es el número de individuos con el que empieza cada especie.
  - Valor por defecto para los **depredadores**: 200
  - Valor por defecto para las **presas**: 12

En cuanto a los parámetros que controlan el comportamiento de las plantas se tiene:

- **Ratio de crecimiento de las plantas** es un multiplicador que se aplica al final de cada etapa de evolución a la población de las plantas. Valor por defecto: 1,7
- **Treshold** es la cantidad mínima de plantas que puede haber, su población nunca será menor a este parámetro. Valor por defecto: 200

- **Población inicial** con la que se comienza la simulación. Valor por defecto: 1200

Por último tenemos la selección de estrategia:

- **Metaheurística/Estrategia** es el método que utilizarán los depredadores para calcular su posición en cada iteración. Por defecto se selecciona Estrategia simple.
  - **Estrategia simple**: los depredadores cazan de manera individual.
  - **Particle Swarm Optimization**: los depredadores cazan en manada actualizando su posición según el algoritmo *PSO*
  - **Grey Wolf Strategy**: los depredadores cazan en manada actualizando su posición según el algoritmo *GWO*
  - **Grey Wolf Strategy**: los depredadores cazan en manada actualizando su posición según el algoritmo *WOA*
- **Iteraciones** que se ejecutarán con cada algoritmo. Si se ha elegido la Estrategia Simple no es necesario indicar ningún valor aquí. Valor por defecto: 200
- **Agentes de búsqueda** que emplea la metaheurística. Si se ha elegido la Estrategia Simple no es necesario indicasr ningún valor aguí. Valor por defecto: 4

Al pulsar el botón *Comenzar* se ejecutará la simulación con los parámetros definidos. Los cubos azules representan a las presas, cuando llegan a la zona segura (cuando su posición de la coordenada x es mayor que 400 unidades) se detienen. Los cubos rojos representan a los depredadores y seguirán a las presas hasta la zona segura, si cazan a una presa esta se quedará parada.

Bibliografía

[1] J. Nutaro, *Building Software for Simulation: Theory and algorithms, with applications in C++*. Wiley, 2010.

[2] G. W. Flake, *The Computational Beauty of Nature*, ch. 12.1. The MIT Press, 1998.

[3] A. Sharov, "Lotka-Volterra Model." `https://web.ma.utexas.edu/users/davis/375/popecol/lec10/lotka.html`. Último acceso: 04/11/2020.

[4] G. W. Flake, *The Computational Beauty of Nature*, ch. 1. The MIT Press, 1998.

[5] J. T. Francesco Berto, "Cellular Automata." `https://plato.stanford.edu/entries/cellular-automata/#BasiDefi`. Último acceso: 06/11/2020.

[6] S. Wolfram, *Four Classes of Behavior*, ch. 6.2. Wolfram Media. Inc, 2002.

[7] E. F. Codd, *Cellular Automata*. Academic Press, 2014.

[8] G. W. Flake, *The Computational Beauty of Nature*, ch. 12.4. The MIT Press, 1998.

[9] D. Tyler, "T*he power of videogame engines.*" `https://www.gamedesigning.org/career/video-game-engines/`. Último acceso: 19/09/2021.

[10] S. Wickramasekera, "H*ow open starndards are making game engines more accessible for serious simulation applications.*" `https://pitchtechnologies.com/2021/09/open-standards-are-making-game-engines-more`

Último acceso: 19/09/2021.

[11] M. Lewis, "G*ame engines in scientific research.*"
https://www.researchgate.net/publication/278232144_Game_engines_in_scientific_research.
Último acceso: 19/09/2021.

[12] "Desarrollo iterativo e incremental."
https://stringfixer.com/es/Iterative_and_Incremental_Development.
Último acceso: 18/09/2021.

[13] T. L. I. Project, "Graphic engine definition."
http://www.linfo.org/graphic_engine.html.
Último acceso: 01/12/2020.

[14] Unity, "Prefab manual."
https://docs.unity3d.com/Manual/Prefabs.html.
Último acceso: 01/12/2020.

[15] Unity, "Scene manual."
https://docs.unity3d.com/Manual/CreatingScenes.html.
Último acceso: 01/12/2020.

[16] T. Interactive, "The 5 best uses for unity."
https://transforminteractive.com/5-best-uses-for-unity/.
Último acceso: 01/12/2020.

[17] G. for geeks, "Introduction to c#."
https://www.geeksforgeeks.org/introduction-to-c-sharp/.
Último acceso: 01/12/2020.

[18] TIOBE, "Tiobe index for november 2020."
https://www.tiobe.com/tiobe-index/.
Último acceso: 01/12/2020.

[19] Microsoft, "Propiedades (guía de programación c#."
https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/properties#:~:text=Una%20propiedad%20es%20un%20miembro,value%20of%20a%20private%20field.
Último acceso: 03/12/2020.

[20] Microsoft, "Language-integrated query (linq)."
https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/concepts/linq/.
Último acceso: 03/12/2020.

[21] C. E. Cuesta, "Evolución y Adaptación del Software: Principios SOLID.."

[22] M. M. Moreno, "¿Qué son los patrones de diseño?."
https://profile.es/blog/patrones-de-diseno-de-software/.
Último acceso: 18/09/2021.

[23] R. Guru, "Clasificación de patrones."
https://refactoring.guru/es/design-patterns/classification.
Último acceso: 18/09/2021.

[24] R. Guru, "Patrón Builder."
https://refactoring.guru/es/design-patterns/builder.
Último acceso: 18/09/2021.

[25] R. Guru, "Patrón Factory Method."
https://refactoring.guru/es/design-patterns/factory-method.
Último acceso: 18/09/2021.

[26] G. G. Peña, "Metodología de la programación: Patrones arquitectónicos.."

[27] R. D. Hernandez, "The Model View Controller Pattern."
https://www.freecodecamp.org/news/the-model-view-controller-pattern-mvc-architect

Último acceso: 18/09/2021.

[28] R. Guru, "Patrón Composite."
https://refactoring.guru/es/design-patterns/composite.
Último acceso: 18/09/2021.

[29] R. Guru, "Patrón State."
https://refactoring.guru/es/design-patterns/state.
Último acceso: 18/09/2021.

[30] R. Nystrom, *Spatial Partition*, ch. 6. 2014.

[31] R. Guru, "Patrón Strategy."
https://refactoring.guru/es/design-patterns/strategy.
Último acceso: 18/09/2021.

[32] R. Guru, "Patrón Mediator."
https://refactoring.guru/es/design-patterns/mediator.
Último acceso: 18/09/2021.