



**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA**

**GRADO EN INGENIERÍA INFORMÁTICA**

**Curso Académico 2023/2024**

**Trabajo Fin de Grado**

**JUEGO 4x BASADO EN AGENTES**

**HERRAMIENTA PARA LA OBSERVACIÓN Y EVALUACIÓN DE  
COMPORTAMIENTOS EMERGENTES**

**Autor:** David López Pereira

**Tutor:** Gustavo Recio Isasi

**Cotutor:** Jonathan Crespo Herrero



## Resumen

Los sistemas basados en agentes (ABM) son una tendencia creciente a la hora de simular sistemas complejos en múltiples áreas de investigación. Por ello, el objetivo principal de este trabajo es elaborar un sistema basado en agentes desde cero para observar y estudiar los diferentes comportamientos que surgen de la interacción de los agentes en un entorno basado en las restricciones de los juegos de estrategia 4x, así como entender la importancia de estos.

El presente trabajo de fin de grado comienza con un estudio sobre el uso de ABM a través de artículos científicos en biología, economía y juegos, evaluando sus principales ventajas y desventajas. Además, se analizan las implementaciones de código abierto en Python para desarrollar ABM, seleccionando entre ellas el framework Mesa. Posteriormente para explicar el juego 4x desarrollado, se indican en detalle las reglas presentes, incluyendo en ellas tres comportamientos preestablecidos (Explorer, Chaser y Farmer), la composición estructural y los métodos principales creados para su implementación, destacando entre ellos `changeBehaviour()`, siendo este el método que permite modificar de manera dinámica el comportamiento. Una vez explicado el juego, se procede a simular la aparición de tres comportamientos emergentes en los preestablecidos, estudiando para cada uno de ellos cómo se balancea el sistema ante los nuevos cambios. Además, se prueba el sistema con tres nuevas mutaciones para simular diferentes conductas en los agentes, en las cuales destaca `CustomBehaviour` como la principal mutación que consigue derrotar al comportamiento Explorer. Destaca significativamente un comportamiento emergente que surge en la mutación Friendly, cambiando por completo su forma de actuar y convirtiéndolo en un agente hostil.

## Palabras clave:

- ABM
- Juegos 4x
- Mesa
- Python
- Comportamiento emergente



## Índice

<b>Capítulo 1. Introducción.....</b>	<b>1</b>
<b>1.1. Motivación.....</b>	<b>1</b>
<b>1.2. Objetivos .....</b>	<b>2</b>
<b>Capítulo 2. Metodología de desarrollo .....</b>	<b>3</b>
<b>2.1. Estructura del proyecto .....</b>	<b>4</b>
<b>2.2. Presupuesto del proyecto .....</b>	<b>5</b>
<b>Capítulo 3. Estado del arte .....</b>	<b>7</b>
<b>3.1. Trabajos relevantes con ABM .....</b>	<b>7</b>
<b>3.1.1. Propagación de enfermedades infecciosas .....</b>	<b>7</b>
<b>3.1.2. Optimización en modelos económicos .....</b>	<b>9</b>
<b>3.1.3. ABM en juegos.....</b>	<b>10</b>
<b>3.2. Implementaciones presentes en el mercado .....</b>	<b>11</b>
<b>3.2.1. AgentPy .....</b>	<b>11</b>
<b>3.2.2. Repast4Py.....</b>	<b>12</b>
<b>3.2.3. PyNetLogo.....</b>	<b>12</b>
<b>3.2.4. PyPandora.....</b>	<b>13</b>
<b>3.2.5. Mesa.....</b>	<b>14</b>
<b>Capítulo 4. Diseño .....</b>	<b>17</b>
<b>4.1. Elección de la implementación .....</b>	<b>17</b>
<b>4.2. Reglas del juego .....</b>	<b>18</b>
<b>4.3. Relaciones entre clases .....</b>	<b>22</b>

<b>Capítulo 5. Implementación .....</b>	<b>27</b>
<b>5.1. Elección de lenguaje .....</b>	<b>27</b>
<b>5.2. Implementación detallada .....</b>	<b>28</b>
<b>Capítulo 6. Pruebas y evaluación del sistema .....</b>	<b>39</b>
<b>6.1. Cambio dinámico en la estructura.....</b>	<b>41</b>
<b>6.1.1. Nueva clase con cambio de conducta.....</b>	<b>41</b>
<b>6.1.2. Nuevo atributo utilizado en el comportamiento .....</b>	<b>45</b>
<b>6.1.3. Nuevo método y modificación de atributos.....</b>	<b>49</b>
<b>6.2. Nuevos comportamientos.....</b>	<b>54</b>
<b>6.2.1. Agentes con comportamientos aleatorios (RandomBehaviour).....</b>	<b>55</b>
<b>6.2.2. Agentes con comportamiento óptimo (CustomBehaviour) .....</b>	<b>57</b>
<b>6.2.3. Agentes que comparten sus recursos (Friendly).....</b>	<b>60</b>
<b>Capítulo 7. Conclusión .....</b>	<b>65</b>
<b>7.1. Pasos futuros .....</b>	<b>66</b>
<b>Bibliografía .....</b>	<b>67</b>
<b>Anexo .....</b>	<b>71</b>
<b>Anexo A. Código del proyecto .....</b>	<b>71</b>
<b>Anexo B. Análisis del proyecto .....</b>	<b>71</b>
<b>Anexo C. Ilustraciones de ejecuciones del punto 6.1.....</b>	<b>71</b>
<b>Ejecución de cambio dinámico en Explorer .....</b>	<b>71</b>
<b>Ejecución cambio dinámico en Chaser.....</b>	<b>72</b>
<b>Ejecución cambio dinámico en Farmer.....</b>	<b>74</b>

## Índice de ilustraciones

Ilustración 1. Esquema visual del modelo iterativo incremental .....	3
Ilustración 2. Diagrama de iteraciones del TFG.....	5
Ilustración 3. Comparativa de resultados entre EBM y ABM [5] .....	8
Ilustración 4. Conexión entre Python y NetLogo [24] .....	13
Ilustración 5. Diagrama UML simplificado de Mesa [26] .....	14
Ilustración 6. Representación del agente en la simulación.....	19
Ilustración 7. Leyenda con atributos del agente .....	19
Ilustración 8. Representación del planeta sin ser conquistado .....	19
Ilustración 9. Representación del planeta conquistado .....	19
Ilustración 10. Diagrama resumen de un turno en la simulación .....	22
Ilustración 11. Diagrama de clases simplificado .....	23
Ilustración 12. Zoom a los botones de la interfaz gráfica .....	25
Ilustración 13. Zoom a “Frames per Second” en la interfaz .....	25
Ilustración 14. Zoom a los parámetros de la interfaz .....	26
Ilustración 15. Representación web del sistema completo.....	26
Ilustración 16. Modificación de atributo constante y privado con <code>Java.lang.reflect</code> .....	27
Ilustración 17. Diagrama de flujo para el método <code>maybeFight()</code> .....	29
Ilustración 18. Ecuación de Bellman.....	30
Ilustración 19. Diagrama de flujo del método <code>act()</code> de la clase <code>Behaviour</code> .....	32
Ilustración 20. Diagrama de flujo de <code>step()</code> de la clase <code>Game</code> .....	37
Ilustración 21. Diagrama de flujo de <code>step()</code> de la clase <code>Player</code> .....	37
Ilustración 22. Mensajes escritos por terminal.....	40
Ilustración 23. Estado del sistema al producir cambio en primer Explorer .....	42

Ilustración 24. Chaser (morado) encuentra a agente 1 .....	42
Ilustración 25. Chaser conquista planetas de agente 1 .....	42
Ilustración 26. Leyenda del último turno de DummyExplorer en la ejecución .....	43
Ilustración 27. Diagrama changeBehaviour Agressive .....	45
Ilustración 28. Último turno de agente 2 .....	46
Ilustración 29. Leyenda simplificada de Ilustración 28 .....	46
Ilustración 30. Estado con la modificación dinámica .....	46
Ilustración 31. Leyenda simplificada de Ilustración 30 .....	46
Ilustración 32. Leyenda del turno 199 del agente sin modificar .....	50
Ilustración 33. Leyenda del turno 199 del agente modificado .....	50
Ilustración 34. Sistema original con un único Farmer (lima) .....	52
Ilustración 35. Sistema modificado sin presencia de Explorer .....	52
Ilustración 36. Diagrama de flujo de setRandomPriorities() de la clase RandomBehaviour... ..	56
Ilustración 37. Diagrama de flujo de inputPriorities() de la clase CustomBehaviour.....	58
Ilustración 38. Lista de prioridad de los agentes óptimos .....	59
Ilustración 39. Lista de prioridades de Friendly .....	61
Ilustración 40. Diagrama de flujo de ChangeBehaviour() de la clase Friendly .....	62
Ilustración 41. Leyenda de comportamiento emergente de Friendly .....	62



## Índice de tablas

Tabla 1. Recompensas obtenidas por batalla.....	21
Tabla 2. Costes de las acciones en el juego .....	21
Tabla 3. Acciones para obtener balance positivo en iteración 6.....	34
Tabla 4. Lógica especial para cada comportamiento.....	35
Tabla 5. Resultados de sistema completo .....	40
Tabla 6. Evolución del sistema tras la modificación en primer Explorer.....	42
Tabla 7. Resultados con cambio dinámico en primer Explorer.....	43
Tabla 8. Resultados con cambio dinámico en todos los Explorer .....	44
Tabla 9. Comparación de ejecuciones en el último turno de agente 2 (Chaser).....	46
Tabla 10. Resultados con cambio dinámico en primer Chaser.....	48
Tabla 11. Resultados con cambio dinámico en todos los Chaser .....	48
Tabla 12. Resultados con cambio dinámico en primer Farmer .....	51
Tabla 13. Comparación entre sistema original y sistema con modificación global en Farmer	52
Tabla 14. Resultados con cambio dinámico en todos los Farmer .....	53
Tabla 15. Resultados con comportamiento RandomBehaviour .....	56
Tabla 16. Resultados con comportamiento CustomBehaviour .....	60
Tabla 17. Resultados con comportamiento Friendly .....	63
Tabla 18. Resultados con comportamiento Friendly cada 50 turnos.....	63
Tabla 19. Resultados con comportamiento Friendly cada 200 turnos.....	64
Tabla 20. Comparación de ejecuciones con el sistema original .....	65



## Capítulo 1. Introducción

En los últimos años, los **sistemas basados en agentes** o **ABM** (“Agent-Based Models”) se han convertido en una creciente tendencia en los estudios de ciencias sociales, economía y biología entre muchas otras disciplinas [1].

Para entender por qué son tan utilizados estos sistemas se debe comprender su significado. Aunque no existe una definición establecida, los ABM son modelos computacionales que permiten generar simulaciones avanzadas en entornos acotados para observar comportamientos que presentan varios agentes.

Los agentes son objetos computacionales que se caracterizan por ser autónomos y que su toma de decisiones esté basada en reglas. Su atributo diferencial se basa en que constantemente reciben señales de cómo se comporta el entorno y de qué manera los otros jugadores interactúan con él, por lo que en función de esas señales deciden cuál es el movimiento óptimo para sus intereses, dando lugar a numerosos comportamientos que no estaban contemplados en un inicio, llamados comportamientos emergentes [2].

### 1.1. Motivación

En entornos complejos, como los sistemas sociales, es crucial contar con métodos de análisis que capturen las interacciones individuales entre los agentes, así como los comportamientos emergentes que resulten de las mismas. Los ABM permiten simular y estudiar tanto las relaciones individuales como los cambios espontáneos en los agentes. Además, a diferencia de los modelos tradicionales, los ABM permiten explorar cómo variaciones individuales pueden desencadenar en efectos a nivel del sistema global [2].

El presente trabajo desarrolla una herramienta basada en agentes diseñada desde cero, para comprender todos los aspectos a considerar a la hora de desarrollar un ABM y observar las relaciones individuales entre los agentes, así como estudiar los diferentes comportamientos emergentes que surjan en el sistema.

## 1.2. Objetivos

El objetivo principal de este trabajo se centra en generar un ABM desde cero pensado como una herramienta que permite la creación de distintos comportamientos encapsulados en un juego 4x. En el que sea posible observar y modificar los comportamientos de los agentes a través de una interfaz gráfica y mediante las funcionalidades del framework seleccionado. Con el propósito de encontrar distintas estrategias y comportamientos que reproduzcan conductas óptimas para las reglas propuestas del juego.

Para poder llegar a este objetivo final se deben cumplir una serie de objetivos intermedios que se describen a continuación:

1. Estudiar cómo implementar un ABM. Eligiendo el lenguaje de programación correcto y el framework correspondiente.
2. Crear un sistema de reglas para que el juego se ajuste a la temática de un juego 4x y presente una cierta complejidad, permitiendo que existan diferentes comportamientos en los agentes.
3. Implementar este sistema en el lenguaje de programación elegido.
4. Analizar los diferentes comportamientos presentes en los agentes y los resultados obtenidos ante ciertos estímulos.

## Capítulo 2. Metodología de desarrollo

La metodología implementada para la generación de este proyecto se basa en un modelo iterativo incremental. En esta metodología se puede entender cada iteración como un mini proyecto en el que cada entrega añade valor y completa el trabajo final.

Este modelo es interesante porque permite modificar de una manera simple y rápida el proyecto, además, permite identificar y solucionar de una manera sencilla los errores, debido a que cada iteración del modelo es un proceso que consta de 5 partes (véase Ilustración 1):

1. **Planificación.** Se establecen los principales requisitos y objetivos de la nueva iteración.
2. **Diseño.** En este apartado se define la lógica y se estudia cómo se quieren implementar los requisitos.
3. **Implementación.** Se desarrollan e implementan las nuevas funcionalidades considerando los requisitos anteriores.
4. **Pruebas.** Una vez implementado se deben realizar las comprobaciones necesarias para asegurar que todo funcione correctamente.
5. **Evaluación.** En cada entrega se busca la retroalimentación de los tutores verificando que se hayan cumplido los objetivos planificados, mejorando así la calidad final de la iteración.

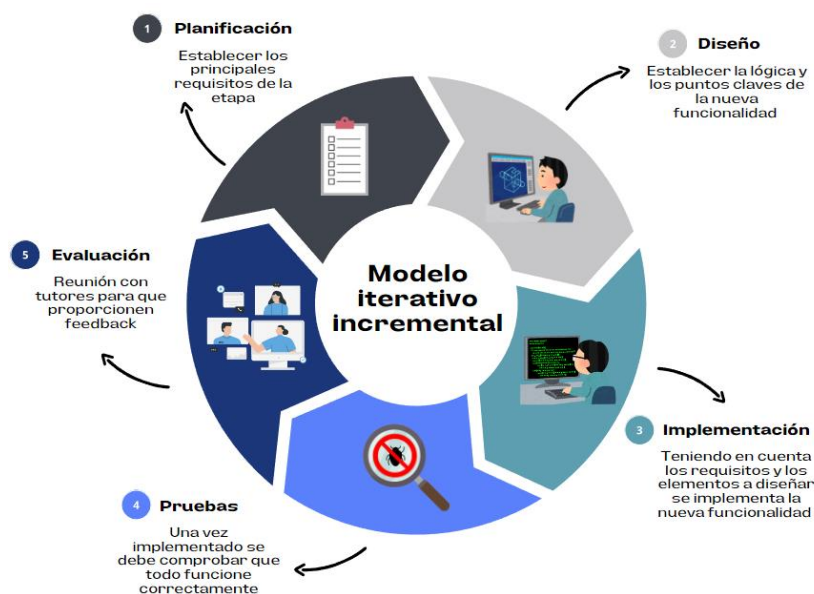


Ilustración 1. Esquema visual del modelo iterativo incremental

## 2.1. Estructura del proyecto

En la Ilustración 2 se observa cómo el proyecto se compone de 8 iteraciones, 7 cortas y una iteración que abarca toda la duración del trabajo, dado que la última se corresponde con la documentación del proyecto, se excluirá de la explicación y solo se tratarán las iteraciones de la primera hasta la séptima. Los nombres presentes en cada iteración representan las acciones realizadas, seguidas del tiempo aproximado que tomó realizarlas.

El desarrollo de este TFG se divide en 8 iteraciones que representan dos grandes bloques, el bloque de investigación y el bloque de implementación.

- **Bloque de investigación.** Abarca todo el conocimiento necesario para poder implementar los ABM, desde entender las diferentes opciones presentes en el mercado que faciliten su implementación, hasta comprender cómo otros investigadores utilizan los agentes para generar comportamientos emergentes. Además, cubre toda la investigación y posterior creación de las reglas del juego 4x. Este bloque representa la iteración 1 en la Ilustración 2.
- **Bloque de implementación.** Representa todos los detalles relevantes a la implementación del ABM basado en un juego 4x, desde la elección del lenguaje hasta la explicación de los algoritmos generados en el proceso. Este bloque abarca desde la iteración 2 hasta la iteración 7 de la Ilustración 2.

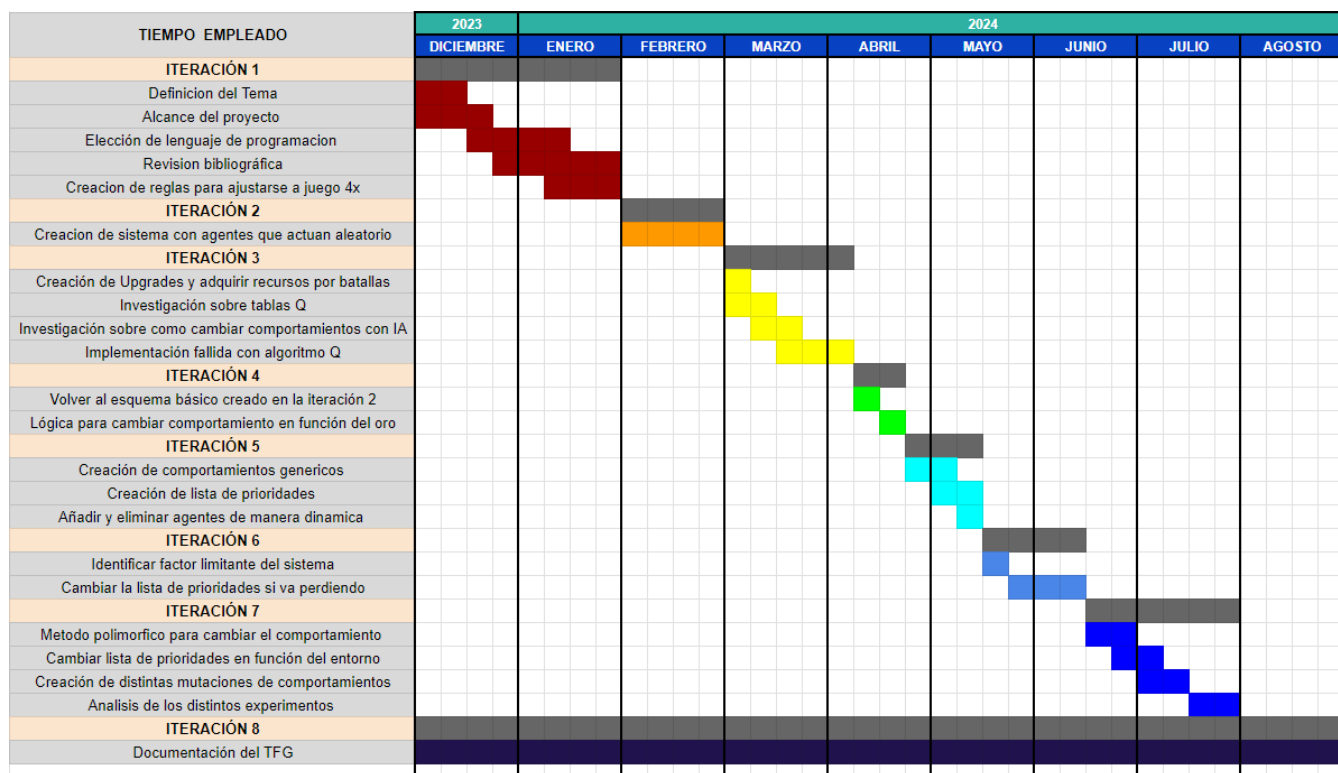


Ilustración 2. Diagrama de iteraciones del TFG

## 2.2. Presupuesto del proyecto

El tiempo invertido en el desarrollo del TFG ha sido de aproximadamente 770 horas distribuidas a lo largo de un año. Con el objetivo de analizar de manera correcta el coste que supondría el proyecto, se considera tanto el pago al personal, como el equipo utilizado para realizar el mismo.

En cuanto al coste del personal, se asume que el alumno estará cobrando un total de 12 euros por hora de trabajo, al ser un estudiante de ingeniería informática sin graduar. Por tanto, el total del coste de personal equivale a:

$$770h \times 12€/h = 9.240 €$$

Todo el TFG se ha desarrollado en un ordenador de sobremesa valorado actualmente en 1000€. El equipo utilizado también engloba el software y las licencias adquiridas para poder realizar este proyecto. Pero, en esta ocasión, no se ha adquirido ninguna de ellas, por lo que los gastos referentes serán 0. Dejando un valor total de equipo equivalente a:

$$1.000€ (\text{ordenador}) + 0€ (\text{licencias adquiridas}) = 1.000€$$

Por último, para calcular el presupuesto total del proyecto se deberá sumar el coste del personal y de equipo.

$$9.240\text{€ (coste personal)} + 1.000\text{€ (coste de equipo)} = 10.240\text{€}$$

Valorando el proyecto final con un coste de 10.240€ en total.



## Capítulo 3. Estado del arte

### 3.1. Trabajos relevantes con ABM

Este apartado sirve a modo de contexto para entender cómo se trata en distintos trabajos la interacción entre agentes y los comportamientos emergentes que surgen de estas interacciones. Aunque no se tratará en profundidad, cabe destacar uno de los primeros trabajos que introdujo el concepto de emergencia en los comportamientos: el Juego de la Vida, creado por el matemático John Conway en 1970 [3], donde por medio de 3 simples reglas las células presentaban comportamientos complejos e impredecibles. Este descubrimiento inspiró las investigaciones futuras en diversos campos científicos, permitiendo tanto el desarrollo de autómatas celulares como de sistemas basados en agentes [4].

#### 3.1.1. Propagación de enfermedades infecciosas

Con el objetivo de entender los beneficios y desventajas del uso de ABM, se presenta el trabajo de Kasareka et al. [5], que compara el uso de sistemas basados en agentes (ABM) con sistemas basados en ecuaciones (EBM), teniendo como ejemplo concreto la propagación del virus de la COVID-19.

Históricamente los sistemas basados en ecuaciones han sido la principal opción a la hora de modelar la propagación de enfermedades debido a su rapidez [6][7][8], sin embargo, no permiten obtener una población heterogénea, pudiendo dejar de lado casos más concretos. Por otro lado, los ABM solucionan este problema proponiendo agentes heterogéneos, pero necesitan una mayor carga de trabajo para inicializar el sistema. Las conclusiones recogidas por Kasareka et al. [5] en la Ilustración 3, muestran cómo las curvas del EBM son más suaves, mientras que el ABM presenta curvas irregulares parecidas a la forma en la que se comportaría la población, gracias a la heterogeneidad y las interacciones de los agentes con su entorno.

Aunque los resultados son más aproximados a la forma real, el coste de tiempo y memoria comparados con los proporcionados por los EBM hacen que sea más interesantes utilizar modelos híbridos que combinen los resultados generales de estos con las interacciones de los ABM.

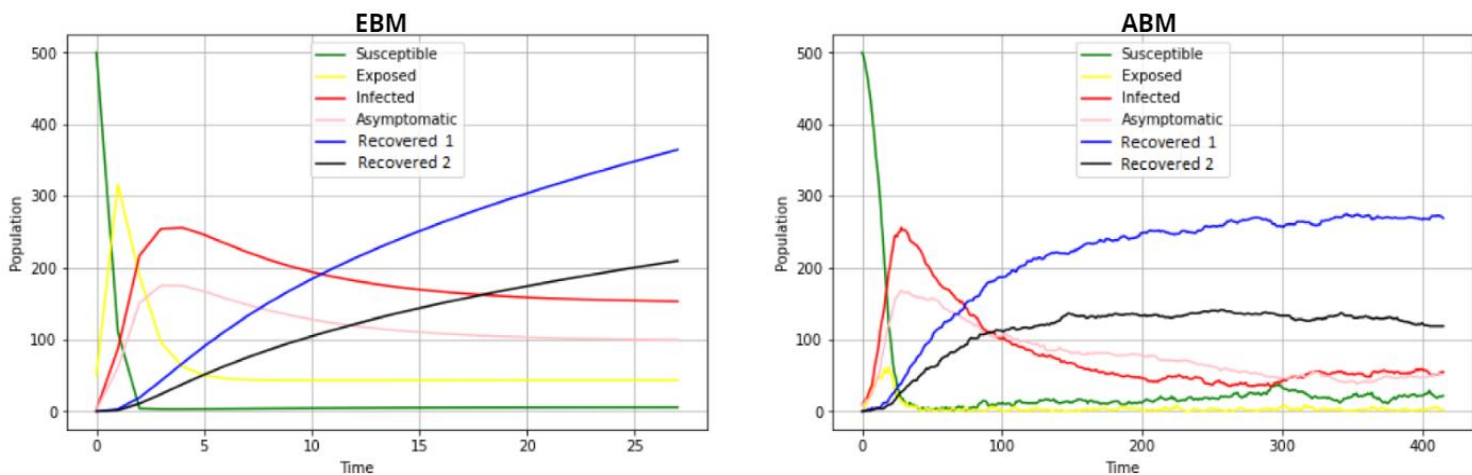


Ilustración 3. Comparativa de resultados entre EBM y ABM [5]

Existen numerosos ejemplos de modelos híbridos utilizados para estudiar la forma en la que las enfermedades infecciosas se propagan [9][10], por ejemplo, el presentado en el artículo científico que simula cómo se realiza la propagación de la COVID-19 en un entorno cerrado de construcción con un gran número de agentes [11]. Explicando de manera detallada el riesgo de infección por medio del modelo matemático de enfermedades infecciosas SEIR, que separa a los agentes en cuatro estados (susceptible, expuesto, infeccioso y recuperado). El ABM es utilizado en los distintos escenarios propuestos para el control de las infecciones permitiendo a los agentes ajustarse a ellos haciendo uso, por ejemplo, de la mascarilla o manteniendo la distancia social, gracias a la heterogeneidad, también se generan escenarios donde haya agentes que cumplan todas las medidas propuestas, estos fueron los que menos riesgo de contagio obtuvieron en los resultados.

Como se ha demostrado en los artículos científicos explicados en este subapartado, las interacciones entre agentes heterogéneos permiten simular de manera precisa los comportamientos de una población sin necesidad de exponerla al riesgo de contagio. Además, la unión con sistemas de ecuaciones, permiten complementar a los ABM. Demostrando que este tipo de modelos pueden desarrollar sistemas complejos con múltiples agentes y añadir diferentes funcionalidades sin comprometer su funcionamiento y mejorando su rendimiento.

### 3.1.2. Optimización en modelos económicos

Para este subapartado se muestra el trabajo de Brusatin et al. [12], basado en la simulación de un entorno económico y cómo afecta a este sistema la incorporación de agentes racionales (óptimos) por medio del uso de ABM y aprendizaje por refuerzo.

Los modelos más usados en las simulaciones económicas están basados en el equilibrio general dinámico estocástico (DSGE) [13] y están lejos de ser una representación fiel. Esto se debe a los problemas de heterogeneidad de los agentes y las suposiciones que emplean, por estos motivos se estudia el impacto en la simulación de un sistema que incorpore un ABM para proporcionar heterogeneidad [12]. Además, con el objetivo de evitar las suposiciones de los agentes del modelo DSGE y que el sistema sea realmente racional incorporan aprendizaje por refuerzo, donde por medio de una “función de utilidad” aprenden de su entorno y mejoran su toma de decisiones.

En los resultados se pueden observar que los agentes entrenados con aprendizaje por refuerzo llevan a cabo estrategias emergentes en función de los niveles de competencia del mercado. Además, a pesar de no comunicarse entre ellos, se observa cómo se agrupan en diferentes grupos estratégicos, aumentando la economía global del grupo. Las estrategias utilizadas para maximizar el beneficio en los agentes son las siguientes:

- Aprovechan la imperfección del mercado, producen y venden pequeños bienes, pero cobran un precio muy alto.
- Maximizan sus beneficios bajando sus precios por debajo del mercado para eliminar competencia y produciendo el máximo posible para aumentar las ventas y ganar cuota del mercado.
- Consiguen beneficios sostenibles y evitan la quiebra mediante estrategias conservadoras que consisten en producir y vender cantidades moderadas de bienes aplicando los precios del mercado.

Este artículo demuestra el potencial que pueden tener los ABM combinados con otras herramientas para encontrar estrategias óptimas en los agentes, observando la colaboración entre ellos para obtener un beneficio mutuo, mejorar la calidad de los sistemas simulados y obtener ideas aplicables en el mundo real.

### 3.1.3. ABM en juegos

Al ser este trabajo un ABM combinado con un juego de estrategia, se debe estudiar y entender de qué manera se aborda esta combinación en los diferentes trabajos publicados. Para ello se muestra el siguiente trabajo de Szczepanska et al. [14], que realiza una revisión sistemática de la literatura (SLR) estudiando las distintas formas de unir los juegos y los ABM dependiendo de las necesidades y los datos que se quieren obtener. La unión de estos dos conceptos (GAM) permite obtener información valiosa para entender las dinámicas sociales complejas. Los juegos, permiten captar la intuición y las decisiones en tiempo real, mientras que los ABM ofrecen una manera de explorar exhaustivamente las dinámicas a lo largo del tiempo y en diferentes escenarios.

Se analizan 52 trabajos científicos publicados entre 2001 y 2020, en ellos se observan 6 enfoques diferentes para la integración del GAM. Además, dado que en el artículo no se comentan en profundidad los ejemplos (solo se mencionan en el apéndice), se añade un ejemplo de los 52 estudiados para cada enfoque con el fin de aumentar la claridad.

1. **ABM como evaluación de juegos.** Utilizan modelos para analizar los resultados de los juegos. Ejemplo [15].
2. **ABM como sustituto de los jugadores en juegos.** Usan agentes en lugar de jugadores reales para entender y comparar diferentes teorías de cooperación. Ejemplo [16].
3. **ABM para estudiar los resultados de los juegos.** Extiende los resultados del juego mediante un ABM que los simula en escenarios grandes a largo plazo. Ejemplo [17].
4. **Juegos para validar ABM.** Usan juegos para verificar que los ABM son precisos. Ejemplo [18].
5. **Juegos híbridos (humanos y agentes juntos).** Esta combinación permite estudiar cómo los humanos responden a las decisiones y comportamientos de los agentes autónomos. Ejemplo [19].
6. **ABM para diseñar juegos.** Utilizan ABM para estructurar y diseñar los juegos. Ejemplo [20].

Asimismo, los autores [14] detallan un esquema para estandarizar la justificación del tipo de combinación entre el juego y el ABM utilizado, dado que en la mayoría de los 52 trabajos existe una falta de información sobre este tema.

Gracias a la popularidad de los juegos y la gran capacidad de los ABM para gestionar la complejidad, se comprende la importancia de combinar ambos en numerosas investigaciones interdisciplinarias. Debido a que, dependiendo de las necesidades del estudio, se puede elegir un enfoque u otro para simular escenarios complejos o estudiar sus conclusiones, lo cual es muy costoso en términos de tiempo sin el uso de los ABM.

### 3.2. Implementaciones presentes en el mercado

En el anterior apartado se ha observado la importancia de los ABM, estos se pueden implementar en los lenguajes de programación más comunes hoy en día, como son Python, Java y C++ [21]. Debido a que los ABM son multidisciplinarios, el uso de frameworks y bibliotecas permite que los desarrolladores se centren en la lógica de los agentes en lugar de detalles de implementación.

En este apartado la investigación se centra en la explicación de los puntos más destacados de 5 implementaciones de código abierto para el lenguaje de programación Python presentes en el mercado, estas son [21]: **AgentPy**, **Repast4Py**, **PyNetLogo**, **pyPandora**, **Mesa**.

#### 3.2.1. AgentPy

AgentPy es una biblioteca de software para el desarrollo y análisis de ABM de manera eficiente y sencilla. Su documentación facilita su uso, proporcionando a los usuarios las herramientas necesarias para crear y analizar modelos simples con facilidad. La biblioteca está optimizada para su integración en entornos de computación interactiva, como Jupyter Notebooks, una aplicación web interactiva que permite ejecutar piezas de código de manera individual. Una de las características más notables es que permite ejecutar simulaciones en paralelo sin la necesidad de escribir código específico para la paralelización [21][22].

Sin embargo, aunque presenta una gran documentación, su comunidad no es tan grande en comparación con otras, lo que dificulta encontrar foros de discusión en los que comparar diferentes implementaciones o aprender del resto de la comunidad. Por otro lado, al generar

modelos complejos, el rendimiento se ve comprometido, además, no presenta facilidad para la personalización ni para la visualización de los datos de forma nativa, por lo que se requieren herramientas externas [22].

### 3.2.2. Repast4Py

Repast4Py se trata de un framework para Python que forma parte del paquete de aplicaciones de Repast, una familia con más de 20 años de desarrollo continuado ampliamente reconocida por su robustez y escalabilidad. Está construido sobre Repast HPC un framework para C++ dirigido a expertos para ser usado en grandes plataformas de sistemas distribuidos de datos y en supercomputadores, por lo que Repast4Py proporciona la habilidad de construir ABM distribuidos que se aprovechan de la paralelización utilizando el protocolo Message Passing Interface (MPI), para la comunicación de sistemas distribuidos, permitiendo desarrollar sistemas complejos, este framework está creado para que sea una forma de introducción más sencilla que Repast HPC para investigadores de todo el mundo [23].

Al ser la última incorporación de la familia Repast, su comunidad es pequeña en comparación al resto de aplicaciones de la familia, lo que dificulta encontrar recursos en línea.

### 3.2.3. PyNetLogo

Se trata de una biblioteca de software para acceder y utilizar NetLogo desde Python, donde a través de métodos se puede cargar modelos, ejecutar comandos y obtener valores de los agentes presentes en el modelo, pudiendo utilizar la interfaz gráfica [24].

NetLogo es un entorno de modelado programable implementado en Java y Scala y es considerada la plataforma estándar para la ejecución de ABM, su gran popularidad viene dada por su gran comunidad que constantemente está desarrollando nuevas funcionalidades y su integración en otros lenguajes de programación, como es el caso de PyNetLogo [21].

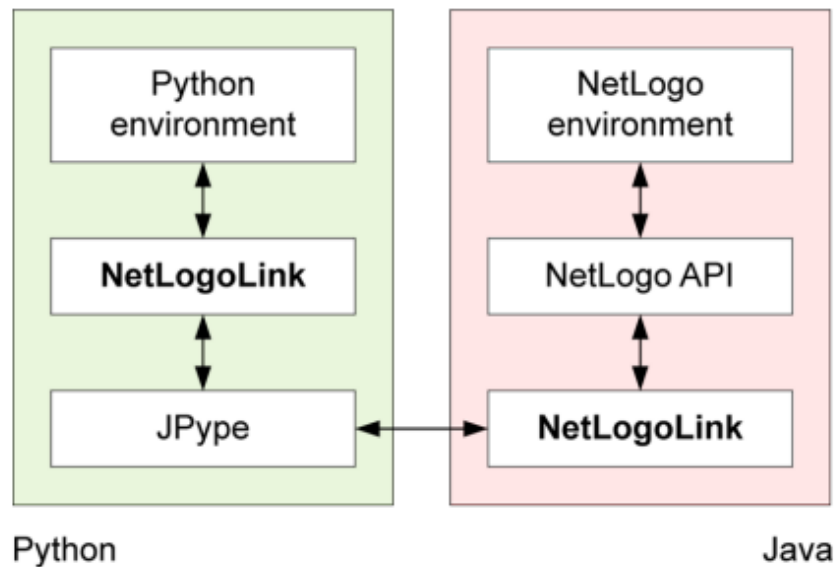


Ilustración 4. Conexión entre Python y NetLogo [24]

PyNetLogo está compuesto por un módulo de Python llamado “core.py” y un archivo Jar llamado “netlogolink.jar”. En la Ilustración 4, se muestra la forma en la que se realiza la conexión entre Python y el entorno NetLogo. Python define la clase NetLogoLink que será utilizada para manejar las interacciones en la parte de Python, gracias a la biblioteca JPype se produce la interacción entre Python y Java. En la parte de Java, el archivo Jar proporciona la clase NetLogoLink para poder manejar las interacciones desde esta parte con la API de NetLogo, permitiendo así la conexión bidireccional entre el código de Python y el modelo de NetLogo en tiempo de ejecución. [24].

### 3.2.4. PyPandora

Pandora surge con la idea de crear un framework que busca cerrar la brecha entre la diversidad de estos modelos existentes para generar ABM, gracias a una interfaz dual en Python y C++, que permite a los usuarios generar tanto prototipos como modelos de computación de alto rendimiento (HPC) sin necesidad de conocimientos en programación paralela [25].

La palabra PyPandora hace referencia a la interfaz desarrollada para Python. Presenta una documentación pobre, sin tutoriales para nuevos usuarios.

### 3.2.5. Mesa

Mesa es un framework modular que permite a los usuarios generar ABM personalizados de una manera simple y rápida, gracias a que múltiples componentes se encuentran ya contruidos. La visualización se realiza por medio de una interfaz web [26].

El término modular significa que el framework se encuentra dividido en el modelado, análisis y la visualización, pero a la hora de ejecutar la aplicación estos módulos trabajarán de manera conjunta.

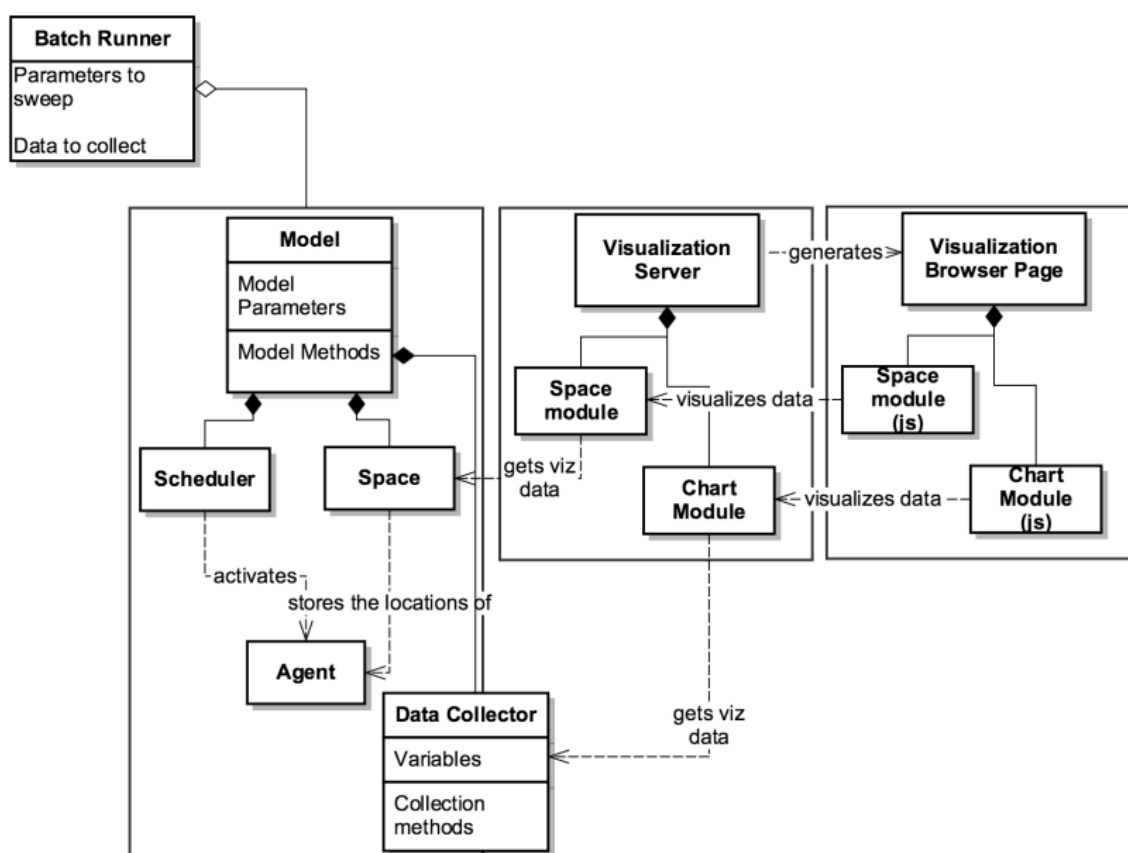


Ilustración 5. Diagrama UML simplificado de Mesa [26]

La Ilustración 5 representa el esquema básico de un modelo de Mesa, mostrando los componentes que lo conforman y la forma en que estos interactúan entre sí. A continuación, se explican en detalle cada uno de los módulos de Mesa junto a sus principales funciones, con el fin de clarificar su funcionamiento [26].



- **Modelado.** Este módulo es el responsable de generar el modelo completo y está compuesto principalmente por cuatro subclases: la clase “**Model**”, que maneja toda la lógica interna del modelo y sirve para almacenar las distintas subclases; “**Agent**”, encargada de definir el funcionamiento de los agentes; la clase “**Scheduler**”, cuya función se basa en determinar en qué momento se ejecutan estos agentes; y la clase “**Space**”, que maneja el espacio donde estarán presentes los agentes. Existen dos tipos de espacios, discretos y continuos, en el discreto los agentes se mueven en casillas y en el continuo cada agente tiene unas coordenadas arbitrarias.
- **Análisis.** Abarca todas las herramientas para recolectar los datos del modelo. La clase “**Data Collector**” es la encargada de guardar los datos relevantes para su posterior análisis con bibliotecas de Python. Además, se hace uso de la clase “**Batch Runner**”, cuya función es permitir la automatización de múltiples ejecuciones y modificar los parámetros del modelo.
- **Visualización.** Con este módulo se pretende presentar el modelo de una manera visual y está separado en dos partes: el servidor “**Visualization Server**” y el cliente que recibe los datos del servidor y utiliza JavaScript para poder mostrarlos en la página web “**Visualization Browser Page**”.



## Capítulo 4. Diseño

### 4.1. Elección de la implementación

Tras estudiar todas las implementaciones en el capítulo anterior, se procede a la elección de la implementación a utilizar. Principalmente los pilares fundamentales sobre los que se basará la decisión serán: la **comunidad/documentación** presente en la implementación, una **curva de aprendizaje poco pronunciada** y la **posibilidad de personalizar** la mayoría de los elementos para ajustarlos de manera sencilla a las necesidades de este proyecto.

- **Comunidad / Documentación.** Mesa destaca por tener una gran comunidad activa y muy presente en foros continuamente generando proyectos [27], solo comparable con la comunidad presente en PyNetLogo. En cuanto a la documentación, Mesa y PyNetLogo vuelven a dominar entre las 5 implementaciones, presentando las documentaciones más extensa y con más tutoriales de esta investigación, destacan negativamente PyPandora por su falta de claridad y AgentPy por su falta de ejemplos en línea.
- **Curva de aprendizaje poco pronunciada.** Para este criterio en la mayoría de las implementaciones la curva de aprendizaje en modelos simples es poco pronunciada, excepto en el caso de Repast4Py que está optimizado para sistemas complejos distribuidos con el protocolo MPI, este factor hace que se descarte esta implementación. Cabe destacar de manera negativa PyNetLogo, que depende de tener instalado NetLogo, lo que incrementa la curva de aprendizaje al requerir el uso de su interfaz.
- **Posibilidad de personalizar los elementos de la implementación.** Al presentar un esquema modular, Mesa permite una personalización superior al resto de implementaciones.

En conclusión, Mesa ha sido la herramienta que mejor se ha adaptado a los criterios fundamentales. Cuenta con una documentación clara y extensa, una comunidad activa, como se puede ver en los numerosos ejemplos en línea [27] y una curva de aprendizaje leve para modelos con un número pequeño de agentes. Gracias a su esquema modular permite modificar en gran parte sus módulos internos obteniendo nuevas funcionalidades. Además, se encuentra en constante desarrollo.

Por tanto, este proyecto se encuentra desarrollado en el framework de Mesa, concretamente en la versión 2.2.4.

## 4.2. Reglas del juego

Antes de establecer las reglas, se debe conocer el significado de este tipo de juegos. Los juegos 4x son un subgénero de los juegos de estrategia en el que el jugador controla un imperio o una raza, con el objetivo de evolucionar, por medio de la obtención de recursos y conquistar nuevos territorios derrotando a los demás jugadores [28]. El término 4x, por lo tanto, hace referencia a la “X” presente en sus cuatro mecánicas principales: exploración, expansión, explotación de recursos y exterminio.

Una vez entendidas las características de este tipo de juegos, se deben generar una serie de reglas que permitan a los agentes poder interactuar con el entorno de una manera suficientemente compleja como para desarrollar comportamientos emergentes.

Este juego está compuesto por **jugadores**, que son los agentes que pueden moverse e interactuar con el entorno y **planetas**, que son la fuente de recursos y proporcionan los puntos estelares necesarios para determinar cuál es el agente ganador, es decir, el agente que mejor se ha adaptado a este sistema y tiene mejor conducta.

Como se ve en la Ilustración 6, el agente se representa por un círculo pequeño que no llena la casilla y al pasar el ratón por encima, se puede ver su información relevante. Además, cada agente está representado por un color diferente, para que sea más fácil distinguirlos. Por otro lado, en la propia interfaz aparece la información relevante a cada agente, como se ve en la Ilustración 7, donde se muestra un pequeño cuadrado con el color del jugador y la información detallada para poder seguir su avance en el juego.

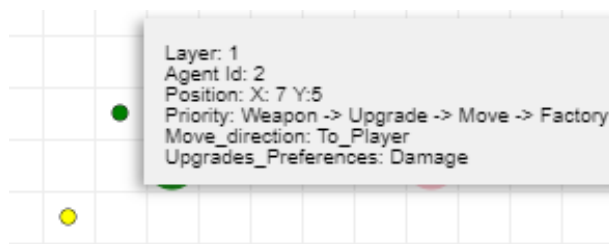


Ilustración 6. Representación del agente en la simulación

■ **Agent: 2** Resources: T: 373 G: 239 P: 2 F: 2 **Stellar Points: 58** Balance: 58  
 Behaviour: **Chaser** Weapon: **Guided missil** Battles Won: 26 Upgrades: **D** ↑

Ilustración 7. Leyenda con atributos del agente

En cuanto a los planetas, estos son representados por círculos grandes que llenan toda la casilla. Si estos no están conquistados, se muestran de color rojo, al pasar el ratón por encima se pueden ver sus recursos y su posición (véase Ilustración 8). Si el planeta ha sido conquistado por un agente, obtendrá el color del jugador conquistador y cambiará su nombre para poder mostrar el Id del agente, en la Ilustración 9 se muestra como el mismo planeta de la Ilustración 8 es conquistado por el agente 2.



Ilustración 8. Representación del planeta sin ser conquistado



Ilustración 9. Representación del planeta conquistado

Una vez definidos los elementos presentes en la simulación, se proceden a explicar las normas establecidas para este juego, que son las siguientes:

- El juego se desarrolla en una plantilla de 20x20 casillas.
- El sistema iniciará con 3 jugadores y 10 planetas.

- La posición de los jugadores es aleatoria, al igual que su color.
- Los planetas son posicionados de manera aleatoria, dejando un espacio entre planetas de 2 celdas en todas las direcciones, para evitar que haya superposición.
- Los agentes, cuentan con 4 atributos principales, oro, tecnología, puntos estelares y un comportamiento.
- Los agentes cuentan con 3 comportamientos preestablecidos. “**Explorer**”, que se dedicará a conquistar planetas y evitar luchas, “**Chaser**”, que se dedicará a perseguir a los demás jugadores para ganarles en duelos y “**Farmer**”, que se dedicará a generar fábricas para obtener recursos.
- Los agentes pueden desarrollar hasta 3 armas diferentes. “**Lasers**” el primer arma que cuenta con 2 de daño, “**Plasma Cannon**” que añade una pequeña mejora de armamento con 3 de daño y, por último, “**Guided missil**” siendo la mejor arma con 5 de daño.
- Los agentes pueden crear fábricas para obtener cada turno 1 unidad de tecnología y 5 de oro por cada fábrica.
- Los que tengan el comportamiento Chaser cuentan con una mejora de daño y los que sean Farmer, con una mejora que dobla los recursos proporcionados por las fábricas.
- Al iniciar los agentes tendrán 30 unidades de tecnología, 100 unidades de oro, 0 puntos estelares, el comportamiento establecido y no contarán con arma ni mejoras.
- Cada planeta tendrá un valor aleatorio de tecnología y oro, en el caso de la tecnología varía entre 0 y 20 unidades y en el caso del oro entre 10 y 50 unidades.
- Los puntos estelares, serán el indicador de cómo se están desarrollando los agentes en el sistema. El agente recibirá tantos puntos estelares como planetas tenga conquistados ese turno, también recibirá puntos estelares por sus duelos ganados.
- Si un agente no logra mantener puntos estelares positivos en 100 turnos, será eliminado de la simulación.
- Cada 100 turnos se unirá un nuevo agente a la simulación.
- La movilidad de los agentes sigue la vecindad de Moore, es decir, se podrá mover en el conjunto de las 8 celdas que rodean su posición.
- Los agentes podrán conquistar los planetas cuando se encuentren a 1 casilla de distancia al planeta, teniendo en cuenta la vecindad de Moore (diagonales).
- Una vez conquistado el planeta, el jugador recibirá cada turno la tecnología y oro de este.

- El agente tendrá que pagar 25 de oro por cada planeta que tenga conquistado.
- Si el agente se queda sin oro y no puede pagar los 25 de oro por planeta, perderá todos los que haya conquistado previamente.
- Los agentes podrán luchar entre ellos, siempre que alguno de ellos tenga un arma y se encuentren a 1 casilla de distancia, teniendo en cuenta las diagonales.
- Si un agente posee un planeta, al perder el duelo perderá a su vez el planeta conquistado. Si posee más de uno se elegirá de manera aleatoria el eliminado.
- El ganador de la batalla será determinado por el agente que obtenga un número mayor, este es obtenido por un valor aleatorio de 1 a 20 multiplicado por el valor del arma que tenga, si el agente cuenta con mejora de daño se añadirá al número obtenido.
- Para ilustrar las recompensas obtenidas por el ganador se utiliza la Tabla 1.

Tabla 1. Recompensas obtenidas por batalla

ARMA	TECNOLOGÍA	ORO
Los dos agentes cuentan con armas.	5% <sup>1</sup> de la tecnología del perdedor.	10% <sup>1</sup> del oro del perdedor.
Uno de ellos no tiene arma	5 unidades de tecnología	10 unidades de oro

<sup>1</sup> Los porcentajes serán redondeados para evitar los decimales.

- Las acciones de los agentes tienen cierto coste. Para explicar todos los costes se utiliza la Tabla 2.

Tabla 2. Costes de las acciones en el juego

ACCIÓN	TECNOLOGÍA	ORO
Movimiento <sup>2</sup>	5 unidades	10 unidades
Fábrica	(1.2 * número de fábricas) * 5 unidades	(1.2 * número de fábricas) * 30 unidades
Arma	20 unidades	40 unidades
Mejora de daño	100 unidades	500 unidades
Mejora de recursos	200 unidades	1000 unidades

<sup>2</sup> Una vez pagado, el movimiento no tendrá coste.

A modo de resumen se presenta el diagrama de la Ilustración 10 que muestra las posibles maneras de actuar de un agente en un turno, cada acción tendrá los costes de la Tabla 2.

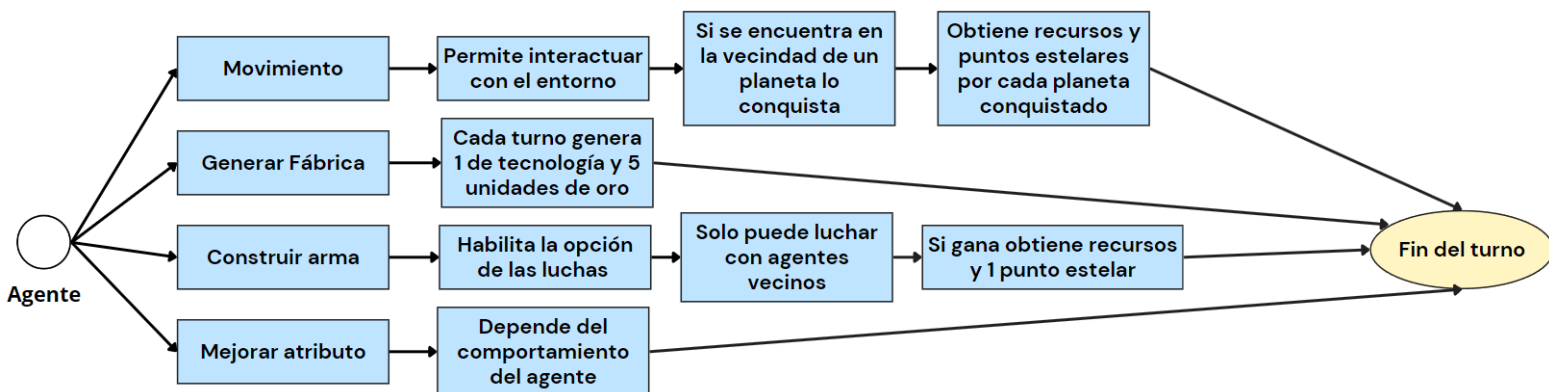


Ilustración 10. Diagrama resumen de un turno en la simulación

Con estas reglas se obtiene un sistema bastante compacto que permite tener un entorno lo suficientemente complejo como para presentar comportamientos diversos. Además, se han cumplido las restricciones para que el juego sea considerado un juego 4x.

La **fase de exploración** se cumple al principio, cuando los agentes se encuentran en posiciones aleatorias y deben avanzar por el tablero para descubrir las posiciones de los distintos jugadores y planetas. Por otro lado, la **parte de expansión** se cumple a la hora de conquistar los planetas evitando al resto conquistarlos. En cuanto a la **explotación**, se puede observar cada turno, porque los agentes que controlan los planetas obtienen la tecnología y el oro de estos, además de ganar puntos estelares. Finalmente, el **exterminio** se lleva a cabo durante los enfrentamientos con el resto de los jugadores de la simulación.

### 4.3. Relaciones entre clases

En este subapartado se pretende explicar de una manera detallada la forma en la que se relacionan entre sí las diferentes clases presentes en el proyecto y las decisiones de diseño tomadas en cada una de ellas.



Para este proyecto se ha adoptado el diseño orientado a objetos que permite tener un entorno estructurado, reutilizable y eficiente para promover la escalabilidad del sistema, gracias al cumplimiento de sus 4 pilares básicos, la abstracción, la encapsulación, la herencia y el polimorfismo.

La Ilustración 11 representa el diagrama de clases UML simplificado, se ha decidido mostrarlo de esta manera, debido a su gran tamaño. Gracias a este diagrama se pueden observar las distintas relaciones entre las clases de una manera clara y sencilla. A continuación, se explican cada una de las clases presentes en el diagrama UML.

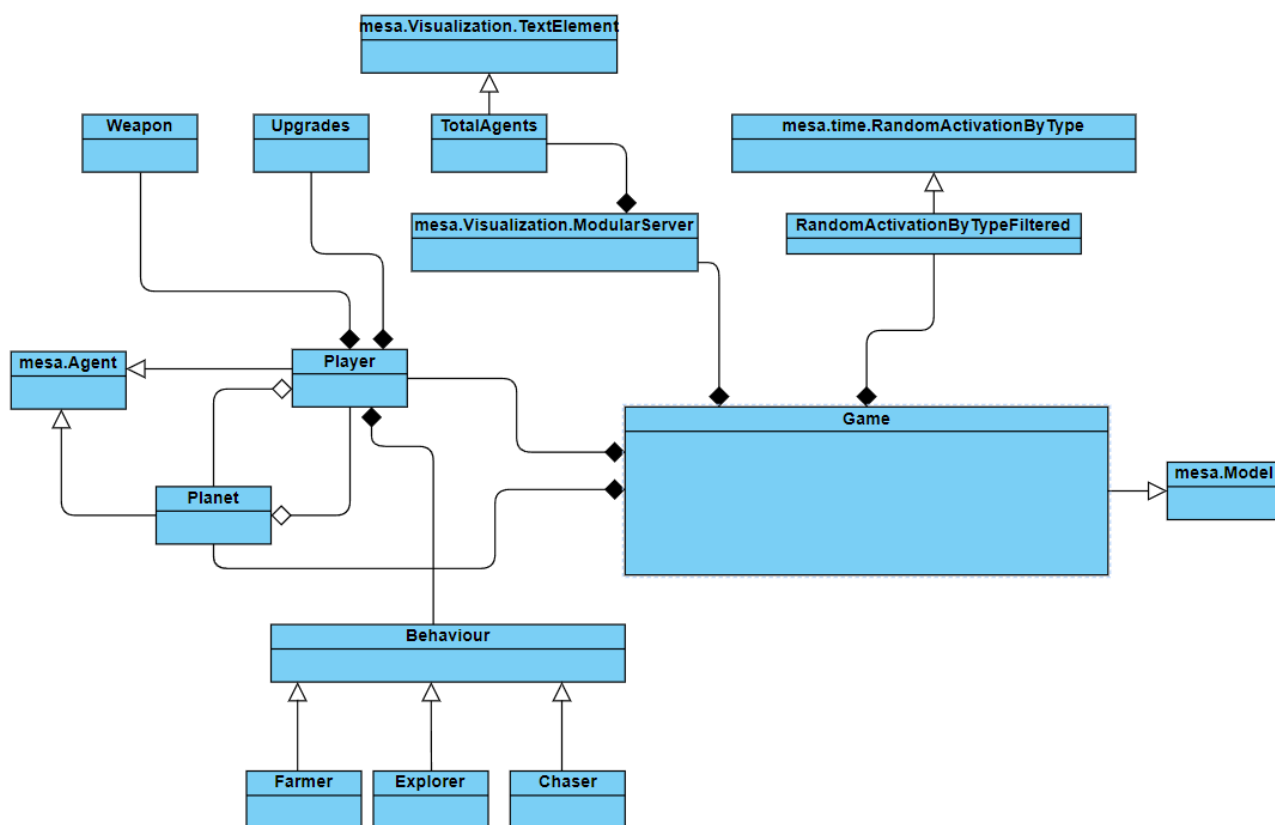


Ilustración 11. Diagrama de clases simplificado

La primera para comentar es la clase **Planet**, esta es la encargada de generar toda la lógica interna para el funcionamiento de los planetas. Hereda de la clase **Mesa.Agent**, debido a que, cada turno debe realizar comprobaciones para saber si algún jugador se encuentra entre sus casillas colindantes. Por lo que, técnicamente, los planetas también son agentes que participan en el sistema, aunque no tienen la posibilidad de moverse. Por motivos de simplicidad, cuando

se nombre a los agentes serán solo a las instancias de la clase **Player**. Además, es la encargada de proporcionar los recursos y deshabitar a los agentes que tengan oro negativo o hayan sido eliminados de la ejecución, es por estos dos motivos que tiene una relación de agregación débil (rombo blanco) con la clase **Player**. También muestra una relación de agregación fuerte (rombo negro) con la clase **Game**, ya que sin el modelo del juego no pueden existir los planetas.

La clase **Weapon** es la encargada de permitir la creación y manipulación de las armas por los agentes, por ello presenta una relación de agregación fuerte con la clase **Player** y es que si desaparece el jugador no puede haber armas en el sistema.

Al igual que la clase **Weapon**, la clase **Upgrades** es simple y presenta una relación de agregación fuerte con la clase **Player**. **Upgrades** se encarga de proporcionar las diferentes mejoras a los jugadores, por lo que, si no hay jugador, no pueden existir mejoras.

En cuanto a los comportamientos, la clase **Behaviour** es la clase padre de todos ellos y se puede entender como el esqueleto que permite la creación de nuevos comportamientos de una manera sencilla, gracias a la herencia. Cabe destacar que presenta una relación de agregación fuerte con la clase **Player**, debido a que sin agente no puede existir ningún comportamiento.

Las clases **Farmer**, **Explorer**, **Chaser**, representan los hijos que heredan de la clase **Behaviour** y son los distintos tipos de comportamiento que podrán tener los agentes. En el caso de querer generar otro comportamiento, bastará con generar una nueva clase que herede de **Behaviour** y cambiar sus atributos principales.

Por el momento todas las clases comentadas presentan una relación de agregación tanto débil o fuerte con la clase **Player**. Esta clase concentra toda la lógica de los jugadores que interactúan con el entorno, se centra en controlar la lucha con otros agentes, realizar la acción seleccionada, administrar los recursos ganados por las fábricas y pagar los impuestos por los planetas conquistados. Este factor hace que la clase presente una agregación débil con **Planet**. Además, al ser un agente en la simulación, hereda de **Mesa.Agent** y se relaciona de forma fuerte con **Game**, ya que sin juego no pueden existir jugadores.

**RandomActivationByTypeFiltered**, es una clase que hereda de **RandomActivationByType**. Su principal función es controlar en qué orden actúa cada tipo de agentes en la simulación. Además, cuenta con el método polimórfico **get\_type\_count()** que será el modificado en la clase

`RandomActivationByTypeFiltered` para obtener los agentes filtrando por el resultado de una función [29]. Presenta una agregación fuerte con la clase `Game`.

**Game** representa la clase más importante del proyecto, debido a que contiene toda la lógica necesaria para el funcionamiento del modelo del juego 4x. Por ello, todas las clases explicadas anteriormente se relacionan fuertemente con ella. Principalmente `Game` proporciona el entorno donde los agentes pueden relacionarse entre ellos, también controla el orden de ejecución de los agentes, además de gestionar la eliminación de los jugadores y la inserción de estos. Al ser el modelo principal del sistema hereda de **Mesa.Model**.

La clase **TotalAgents** sirve para presentar el número total de agentes presentes en la simulación. Hereda de **Mesa.Visualization.TextElement** y presenta una agregación fuerte con **ModularServer**, esta clase sirve a modo de interfaz gráfica basada en web mostrando la plantilla de 20x20, donde se desarrolla el juego, también presenta una gráfica de barras para poder observar un desglose del número de agentes según su comportamiento. Por último, muestra una leyenda con todos los datos relevantes a los agentes. Al ser la forma de visualización del modelo presenta una agregación fuerte con la clase `Game`.

El diseño final de la interfaz gráfica se presenta en la Ilustración 15. Esta interfaz cuenta con unos pequeños botones en la parte superior izquierda que permiten tener diferentes funcionalidades: ejecutar el modelo continuamente mediante el botón “**Start**”, ejecutar paso a paso a través del botón “**Step**” y reiniciar la ejecución para confirmar los cambios de parámetros con el botón “**Reset**” (véase Ilustración 12). La velocidad de la ejecución continua viene determinada por la barra “Frames per Second” (véase Ilustración 13).

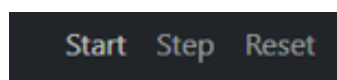


Ilustración 12. Zoom a los botones de la interfaz gráfica



Ilustración 13. Zoom a “Frames per Second” en la interfaz

Además, cuenta con barras laterales para modificar sus parámetros (Ilustración 14), pudiendo simular ejecuciones que comiencen con un número variable de 1 a 6 agentes o 2 a 20 planetas, también presenta una barra que permite modificar el impuesto que se paga por planeta conquistado. Si se modifica algún parámetro se debe pulsar “Reset”, para confirmar los cambios.

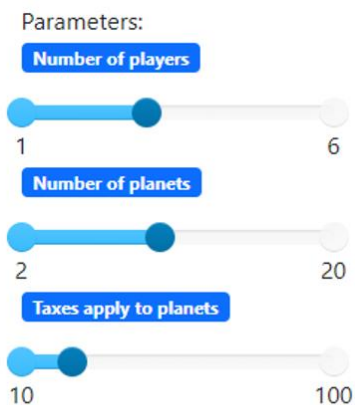


Ilustración 14. Zoom a los parámetros de la interfaz

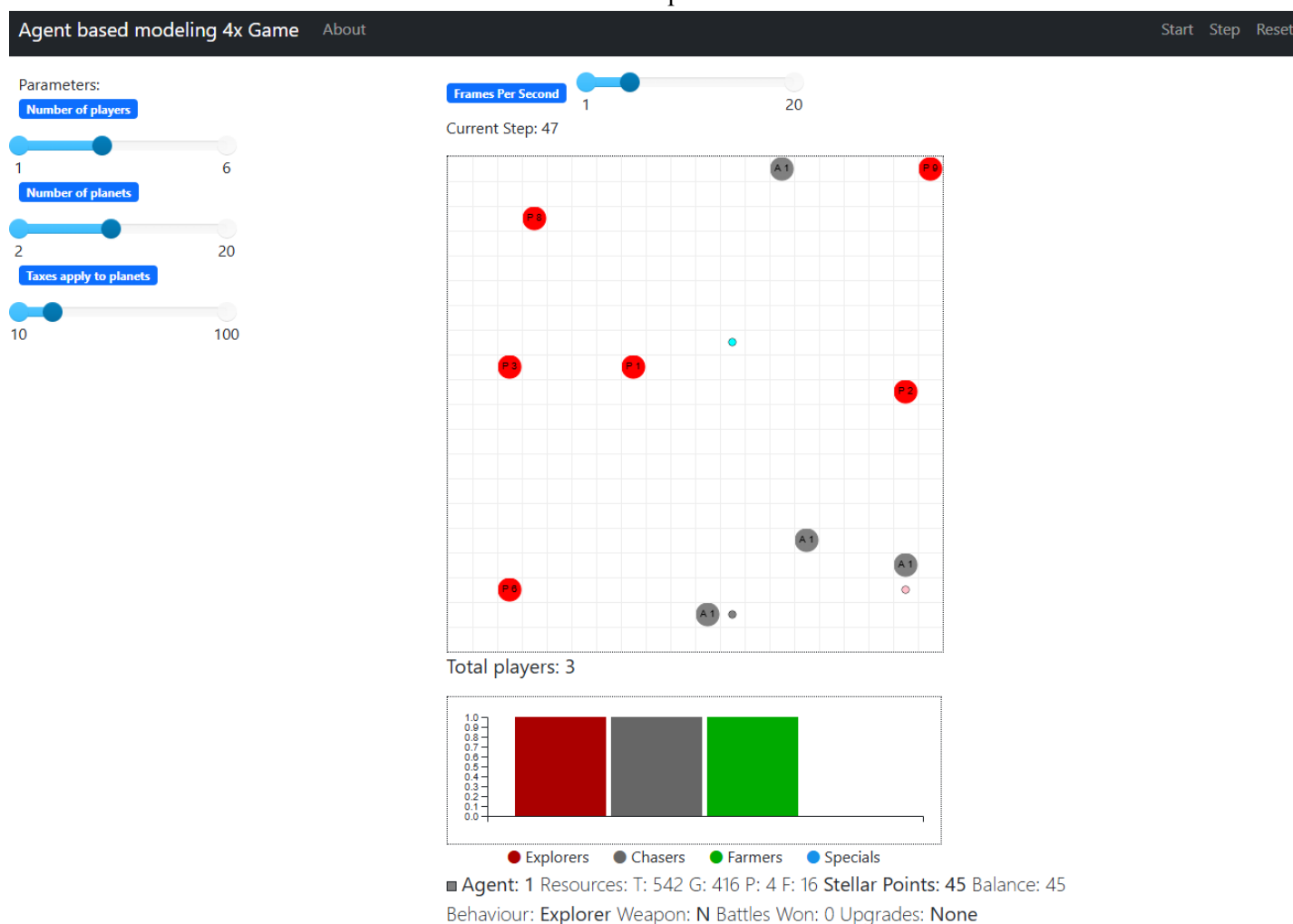


Ilustración 15. Representación web del sistema completo

## Capítulo 5. Implementación

### 5.1. Elección de lenguaje

Inicialmente se consideró la opción de programar el sistema en Java debido a que es un lenguaje fuertemente orientado a objetos, lo que proporciona una estructura de desarrollo robusta y organizada. Sin embargo, uno de los requisitos clave del proyecto es la capacidad de modificar en tiempo de ejecución los comportamientos creados y permitir que los agentes puedan mutar dinámicamente.

Para intentar solucionar este problema se exploró la posibilidad de utilizar la biblioteca `Java.lang.reflect`, esta permite inspeccionar la estructura y manipular el comportamiento de clases, interfaces, métodos y variables en tiempo de ejecución. Además, permite poder acceder a campos privados y poder crear instancias de clases.

Sin embargo, `Java.lang.reflect` tiene 2 grandes desventajas que hacen desechar la idea de utilizar Java como lenguaje para este proyecto.

1. La seguridad del proyecto queda comprometida, rompiendo los principios de encapsulación permitiendo modificar atributos y métodos privados (véase Ilustración 16).
2. El rendimiento es más lento, debido al coste que implica la búsqueda y manipulación en tiempo de ejecución.

A continuación, se muestra un ejemplo de un programa que permite acceder y modificar una variable constante y privada. Para poder ilustrar las desventajas comentadas anteriormente.

```
public static void main(String[] args) throws Exception {
    Persona person = new Persona("David", 23);
    Field[] personFields = person.getClass().getDeclaredFields();
    Method[] personMethods = person.getClass().getDeclaredMethods();

    // Se obtiene el nombre de todas las variables presentes en la clase Persona y cambiamos el nombre que es final y privado
    for (Field field : personFields) {
        System.out.println(field.getName());
        if (field.getName().equals("name")) {
            field.setAccessible(true); // Se hace accesible para que pueda "romper" su condicion de privado y final
            field.set(person, "Edu");
        }
    }
}
```

Ilustración 16. Modificación de atributo constante y privado con `Java.lang.reflect`

El código de la Ilustración 16 recorre por medio de un bucle for todos los atributos de la clase Persona, comprobando si es igual a “**name**” (atributo privado de la clase). Si la comprobación es positiva hace accesible el atributo con el método **setAccessible(true)** y cambia el valor al deseado con **set()**, indicando la instancia de la clase y el nuevo valor.

Con la decisión de no utilizar Java, se decidió buscar lenguajes interpretados [30] para facilitar la inclusión dinámica de código. A diferencia de los lenguajes compilados, los lenguajes interpretados no requieren una compilación previa, en su lugar, ejecutan el programa línea por línea, lo que permite realizar modificaciones en tiempo de ejecución de una manera más ágil.

Algunos de los lenguajes de programación interpretados más comunes son PHP, Python, Ruby y JavaScript. Por tanto, se decidió utilizar Python debido a que, de los lenguajes interpretados mencionados, es el más tratado en la carrera y donde existe un mayor número de implementaciones para ABM, como se ha comentado en el capítulo 3.2.

## 5.2. Implementación detallada

El objetivo de este apartado se centra en dar una idea principal de todos los cambios llevados a cabo en el bloque de implementación y la forma en la que el proyecto evolucionó a lo largo del tiempo, hasta obtener la versión final. Para ello se comentan los puntos principales de las iteraciones comentados en el capítulo de la metodología. Gracias al bloque de investigación, explicado anteriormente en el capítulo de diseño, se obtienen las bases del proyecto sobre el que construir la implementación.

Con la **segunda iteración** comienza el bloque de implementación, donde en la primera versión del programa se generó un esquema básico en el que se tuvieran en cuenta las principales reglas del sistema, el juego debía tener jugadores y planetas en una plantilla de 20x20. Además, los agentes debían poder realizar las 3 acciones básicas, que son moverse, construir fábricas o construir armas, con el objetivo de poder enfrentarse y ganar a los demás jugadores. Los agentes realizaban sus acciones en base a porcentajes, teniendo el 60% el movimiento, 30% la creación de fábricas y el 10% restante para la creación de armas.

Para la implementación de este esquema se utiliza el framework Mesa, gracias a este marco de trabajo se crearon de una manera rápida los primeros pasos de las clases Planet, Player y Game. Además, se generó la clase Weapon por completo. **Weapon** es una clase simple que cuenta con **un diccionario** que tiene como claves el nombre del arma y como valor el número de daño de esta. Además, cuenta con **una lista con las claves del diccionario** para saber el arma actual del agente y si puede realizar más mejoras de armamento. Por lo que, si en un futuro se quiere añadir una nueva arma se deberá introducir una nueva entrada en el diccionario con el nombre y el valor del arma.

Con el esquema creado de la iteración 2, en la iteración siguiente se debían añadir las recompensas económicas de la Tabla 1, además de la última acción a realizar, que era la generación de mejoras. Debido a que se añadió una nueva acción, también se debía modificar la forma en la que seleccionaban las acciones los agentes, eliminando la idea de la utilización de porcentajes, ya que, en la evaluación de la anterior iteración, se observaba cómo todos los agentes realizaban la acción más probable.

Para la implementación de la **iteración 3** se decidió separarla en partes. Se añadieron las recompensas económicas en el método **maybeFight()** de la clase Player, para explicar el funcionamiento de este método se utiliza el diagrama de flujo presente en la Ilustración 17.

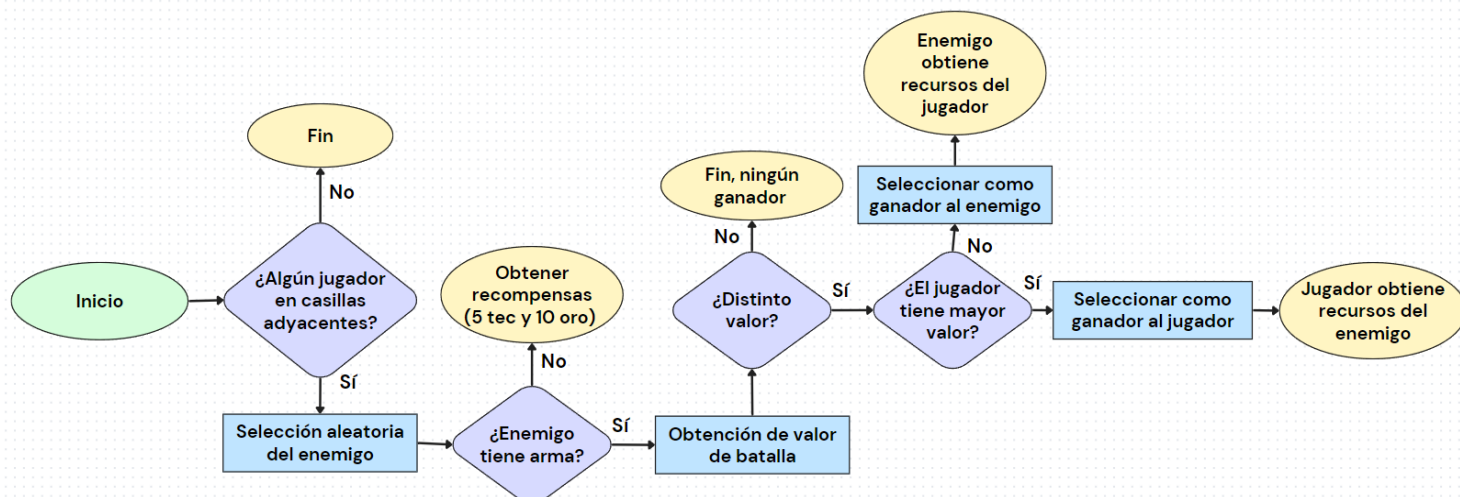


Ilustración 17. Diagrama de flujo para el método maybeFight()

En el caso de la adición de la última acción a realizar se creó la clase **Upgrades**, donde por medio de **booleanos** determina si se pueden realizar las mejoras o ya han sido aplicadas. Además, cuenta con una **lista con las mejoras disponibles** y un número entero para determinar el total de mejoras que ha realizado. Una vez se haya mejorado algún atributo aparecerá en la leyenda del agente con la inicial y una flecha ascendente.

A la hora de decidir la acción elegida por el agente se decidió investigar la adición de aprendizaje por refuerzo en el sistema, por medio del algoritmo de Q-Learning [31]. Este algoritmo se basa en obtener la acción que mayor recompensas futuras proporcionará al agente, por medio de una tabla de valores Q. Esta tabla asocia cada estado con las acciones posibles y sus respectivas estimaciones sobre la recompensa obtenida, para aprender los valores de la tabla Q se utiliza la ecuación de Bellman (véase la Ilustración 18).

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

Ilustración 18. Ecuación de Bellman

Sin embargo, la implementación del algoritmo Q-Learning no parecía encajar bien con el entorno que se había pensado en la primera iteración, los principales problemas encontrados eran el aumento considerado de complejidad al sistema y la actualización de los valores de la tabla Q.

En este sistema la tabla Q cuenta con **44 posibilidades**, este valor se calcula por medio de la multiplicación entre acciones y estados posibles. Se cuenta con 4 acciones posibles (mover, generar fábricas, generar armas y realizar mejoras), pero la acción de mover se subdivide en 8 direcciones por lo que realmente se tendrán que especificar **11 acciones posibles** y se cuenta con **4 estados posibles** para cada casilla (casilla vacía, casilla con planeta, casilla con agente armado, casilla con agente sin arma). Una vez definida la tabla se tienen que asignar unas “recompensas” a cada acción para poder actualizar los valores de esta por medio de la ecuación de Bellman. El problema radicaba en cómo determinar que el estado de la casilla vacía es negativo o positivo, porque dependerá de la situación individual de cada agente. Si, por ejemplo, el agente decide moverse a una casilla vacía que se aleja de un planeta, pero se acerca a un jugador que no tiene arma, se le debería penalizar por alejarse del planeta y a su vez incentivarle



por recortar la distancia con un enemigo sin arma. Por lo que se deberían tener en cuenta más valores que los que aparecen en la tabla Q, aumentando el tamaño y la complejidad de esta.

Debido al aumento de la complejidad, en la fase de evaluación se decidió descartar el algoritmo para este trabajo y en la siguiente iteración sustituir la decisión de los agentes por condiciones en función del oro de estos. Por ello, la **iteración 4** se basaba en eliminar todas las variables relacionadas con la implementación del algoritmo Q-Learning y en la selección de las acciones en función de su oro.

Para la implementación de esta iteración se modificó el método `step()` de la clase `Game` para ajustarse a las necesidades en función de los recursos de los agentes. Se debía pensar una forma en la que los jugadores subsistieran los primeros turnos, es decir, que pudieran moverse y generar alguna fábrica en estos. Por este motivo, cuando el **oro era igual o inferior a 30**, la acción realizada era un movimiento aleatorio, esto se hacía para permitir que el agente consiguiera oro sin realizar acciones que gastaran más recursos. Cuando su **oro era mayor a 30**, realizaba la acción de generar una fábrica para obtener recursos pasivos. Si el **oro era mayor a 40** unidades, se comprobaba la disponibilidad para generar una nueva arma o realizar una mejora de atributos, priorizando siempre la actualización del arma. Si, por el contrario, no se cumplían los requisitos de las opciones anteriores, se llevaba a cabo un movimiento aleatorio que evitaba al agente permanecer quieto en el sistema sin realizar ninguna acción.

Pero en la fase de evaluación de la iteración se observaba cómo los agentes acababan moviéndose aleatoriamente cuando ya no tenían mejoras disponibles, obteniendo resultados similares a los que se obtenían cuando se determinaba la acción por probabilidades. Además, hasta este momento todas las ejecuciones se realizaban con 3 agentes, por lo que se necesitaba añadir agentes nuevos y eliminar a los que no estuvieran jugando de una manera correcta.

En la **iteración 5** como el sistema ya era compacto y el principal problema era la forma de actuar de los agentes, se decidió incorporar los distintos comportamientos (Explorer, Chaser y Farmer), así como la adición y eliminación de manera dinámica de los jugadores.

A la hora de implementar los comportamientos se creó la clase `Behaviour`, que en lo que a sus atributos se refiere, cuenta con un **diccionario de acciones posibles**, utilizado para especificar la dirección de movimiento y las mejoras disponibles para cada comportamiento. También

consta de **una lista de prioridad**, que, como su propio nombre indica, determina la prioridad con la que los agentes realizan las acciones y **dos variables auxiliares** para determinar el objetivo de la dirección de movimiento. Además, de un booleano para saber si tiene que huir o perseguir a un jugador. La acción por realizar se decide gracias al método **act()** el cual es explicado en la Ilustración 19 por medio de su diagrama de flujo, en este al devolver la cadena de texto “Wait”, el agente no realizará ninguna acción en ese turno.

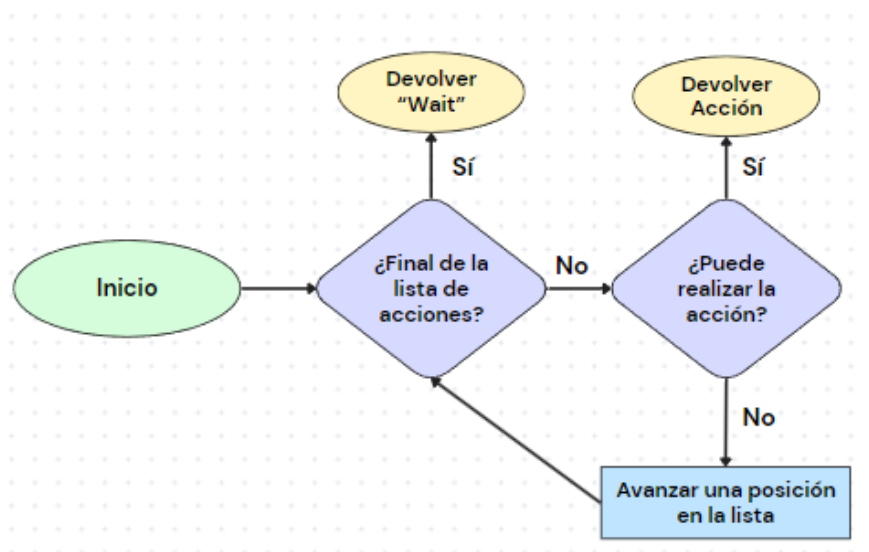


Ilustración 19. Diagrama de flujo del método act() de la clase Behaviour

Gracias a la creación de la clase padre Behaviour, la implementación de los distintos comportamientos es muy simple, tendrán que heredar de Behaviour y modificar su lista de prioridades y su diccionario de acciones para presentar distintas conductas.

En cuanto a la adición de agentes en el sistema se realiza por medio del método **addAgent()**, presente en la clase Game, donde se crea una nueva instancia de la clase Player y se añade al modelo. La selección del comportamiento del nuevo agente se realiza de manera aleatoria entre los 3 preestablecidos.

Para saber que agentes serán eliminados del juego se utiliza la variable correspondiente a su balance de puntos estelares, esta se reiniciará cada 100 turnos y si es negativa borrará al agente de la simulación, esta funcionalidad está implementada en el método **maybeRemoveAgent()**, presente en la clase Game, por medio de un bucle que recorre todos los agentes de la simulación.

Tanto la eliminación como la adición de agentes se realizarán cada 100 turnos, cabe destacar que siempre se añadirán agentes, independientemente de si se ha eliminado o no a un jugador.

En esta iteración también se implementó la funcionalidad para guardar las ejecuciones en la carpeta “*saves*”, esta funcionalidad solo se activa si se ejecuta la carpeta “*model.py*”, donde está presente la clase *Game*. Gracias a esta carpeta se observó en la etapa de evaluación que el sistema siempre se mantenía constante con 4 o 5 agentes, por lo que existía un factor limitante que no permitía que coexistieran más agentes.

La **iteración 6** se centró en encontrar ese factor limitante en el sistema y en intentar implementar un método que realizara siempre las acciones prioritarias para obtener puntos estelares. Este método se pensó principalmente para que los agentes con un balance negativo pudieran olvidarse de su comportamiento y volver a un balance positivo.

El factor limitante del sistema resultó ser debido a varios motivos:

- El primero de ellos era el método *maybeRemoveAgent()* creado en la iteración anterior, ya que, este método eliminaba a todos los agentes con balance negativo sin importar la antigüedad de estos, llegando a eliminar en una misma comprobación hasta 4 jugadores. Este era un sistema que penalizaba mucho a los nuevos agentes que se introducían en la simulación, por lo que se modificó para que al eliminar a un agente se parara la ejecución del bucle y así solo eliminar al más antiguo con balance negativo.
- El segundo factor limitante era el número de planetas presentes en la simulación, al ser solo 10 muchos de los agentes nuevos no tendrían la oportunidad de descubrir nuevos planetas y acabarían con balances negativos, por lo que se introdujo la regla que permitía perder un planeta si se perdía un duelo.
- El último factor limitante era el tamaño de la plantilla, al ser de 20x20 los agentes tendrían una limitación de espacio, por lo que llegados a cierta cantidad siempre se encontrarían con algún agente en sus casillas adyacentes, aumentando las posibilidades de perder duelos y puntos estelares.

Con estas simples modificaciones el número de agentes en la simulación aumentó considerablemente pasando de los 4 o 5 agentes de la iteración anterior, a tener consistentemente 20 en las siguientes ejecuciones.

En cuanto a la implementación de la función para maximizar los puntos estelares, se debía entender la forma en la que se generan dichos puntos y es a través de la conquista de planetas y las batallas ganadas. Una vez entendido se debía hacer que los agentes con un balance negativo olvidarán por completo su comportamiento y se centrarán en realizar las siguientes acciones en función de ciertas restricciones explicadas en Tabla 3.

Tabla 3. Acciones para obtener balance positivo en iteración 6

ARMA	ORO	FABRICAS	ACCIÓN
Sin armas	Negativo	Más de 1 fábrica	Esperar para poder obtener oro y generar armas
Al menos 1 arma	Negativo	Cualquier número	Moverse hacia un agente con peor arma o con mucho oro para luchar contra él
Indiferente	Positivo	Indiferente	Moverse hacia un planeta para conquistarlo

Estas acciones funcionaban en algunas situaciones, pero no parecía ser la mejor opción porque hacía a los agentes comportarse igual, perdiendo su identidad, además muchos de ellos no conseguían volver a un balance positivo, porque perdían las luchas o se quedaban esperando para poder generar algún arma.

Con esta conclusión de no perder la identidad de los agentes para recuperar sus balances comenzaba la **última iteración**, que finaliza el bloque de implementación. En ella se debía realizar una funcionalidad parecida al método para maximizar las acciones, pero individual para cada comportamiento, por lo que se debía entender las motivaciones principales de cada uno de ellos y definir cómo actuarán ante ciertas circunstancias.

Para poder ilustrar la lógica seguida en la implementación se presenta la Tabla 4, donde se explica para cada uno de los comportamientos su lista de prioridad y su forma de actuar ante los 4 estados posibles.

Tabla 4. Lógica especial para cada comportamiento

<b>Explorer. Lista de prioridad: [Moverse → Fábrica → Arma → Mejora]</b>	
<b>Solo planetas adyacentes</b>	Actúa normal, moviéndose hacia los planetas cercanos.
<b>Solo jugadores adyacentes</b>	Si el agente tiene arma, huye de él. Si no tiene ningún arma, actúa normal.
<b>Planetas y jugadores adyacentes</b>	Actúa de la misma manera que si solo hubiera jugadores adyacentes.
<b>Nada en casillas adyacentes</b>	Prioriza la construcción de fábricas para poder subsistir.
<b>Chaser. Lista de prioridad: [Arma → Mejora → Moverse → Fábrica]</b>	
<b>Solo planetas adyacentes</b>	Comprueba si algún planeta tiene más de 25 de oro. Si alguno cumple esta restricción, prioriza el movimiento hacia él. En caso contrario actúa normal.
<b>Solo jugadores adyacentes</b>	Comprueba si los agentes tienen mejor arma que él. Si la tienen, persigue al agente con peor arma en toda la simulación. Si él tiene mejor arma, persigue al que peor tenga de ellos. Si por el contrario no tiene armas, se centra en mejorarlas.
<b>Planetas y jugadores adyacentes</b>	Actúa de la misma manera que lo haría si solo hubiera jugadores adyacentes.
<b>Nada en casillas adyacentes</b>	Si no tiene nada alrededor actúa según su lista de prioridad.
<b>Farmer. Lista de prioridad: [Fábrica → Mejora → Arma → Moverse]</b>	
<b>Solo planetas adyacentes</b>	Cambia su dirección de movimiento dirigiéndose hacia el planeta más cercano, priorizando en su lista el movimiento.
<b>Solo jugadores adyacentes</b>	Comprueba si tiene mejor arma que sus rivales. Si es el caso persigue al que peor tenga. En caso negativo, actúa normal.
<b>Planetas y jugadores adyacentes</b>	Dependerá de si tiene mejor arma que sus rivales adyacentes, si la tiene atacará y si no buscará el planeta cercano.
<b>Nada en casillas adyacentes</b>	Actuará según su lista de prioridad.

Toda esta lógica queda implementada en el método polimórfico **changeBehaviour()** presente en cada una de estas clases. Con la incorporación de este método se debe modificar la forma en la que se produce la selección de acciones por parte del agente, por lo que se explicará de una manera detallada la forma en la que se realiza esta selección.

La llamada de **changeBehaviour()** se realiza en la clase **Player**, gracias al método **selectAction()**, donde primero se obtiene el diccionario con los agentes relevantes con el método **getOtherPlayers()**, en el que las claves serán las siguientes características de los jugadores: mayor puntos estelares, peor arma, mayor número de planetas, más recursos, menos recursos y el planeta que proporciona más oro. El valor de estas claves será el agente correspondiente a cada característica, este diccionario permite dirigir la dirección de movimiento hacía el agente que se quiera sin limitación de encontrarlo en su contexto.

Una vez obtenido el diccionario se separa en listas los tipos de agentes que se encuentran a 2 casillas de distancia del jugador y se pasan como argumentos estos valores al método **changeBehaviour()**. Por último, se declara la acción a realizar con el método **act()** de la clase **Behaviour** que devuelve la cadena de texto correspondiente con la acción a realizar.

Pero, para poder realizar la acción, esta cadena de texto debe ser tratada en la clase **Game**, allí en el método **step()** se transforma el nombre de la acción elegida por el agente en un número que representa cada una de las acciones gracias al método **chooseAction()** y al diccionario **ACTION\_SPACE** presente en el archivo “global\_constants.py”, este diccionario representa números del 0 al 10 para cada posible acción. Una vez tratado se envía como argumento al método **step()** del agente, donde se verifica los requisitos para cada acción y se realiza gracias al método **do\_action()**.

Con el objetivo de poder entender de una manera clara el funcionamiento de las dos clases principales del juego, clase **Player** y clase **Game**, se presenta sus diagrama de flujo correspondientes a sus métodos **step()** en la Ilustración 20 la clase **Game** y en la Ilustración 21 la clase **Player**.

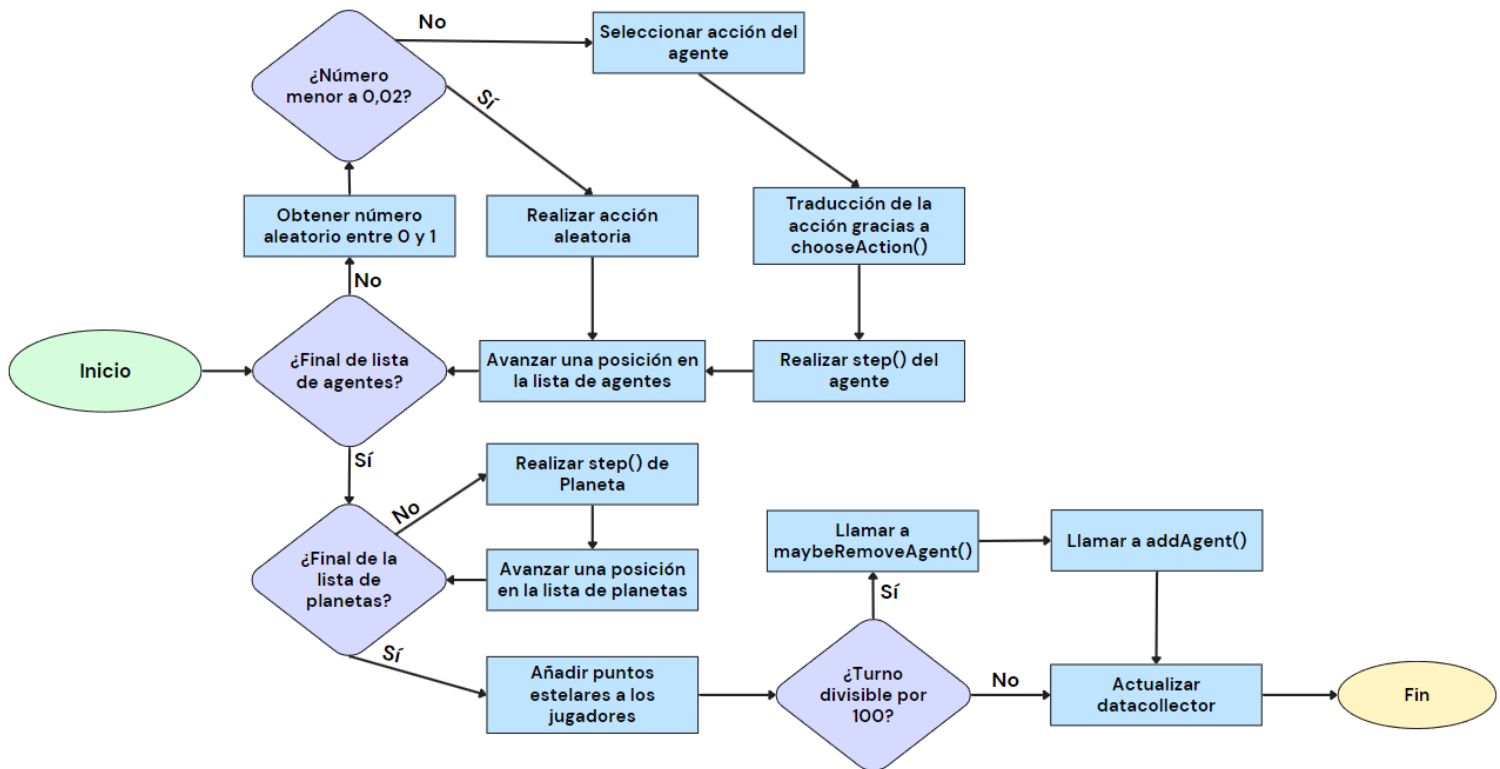


Ilustración 20. Diagrama de flujo de step() de la clase Game

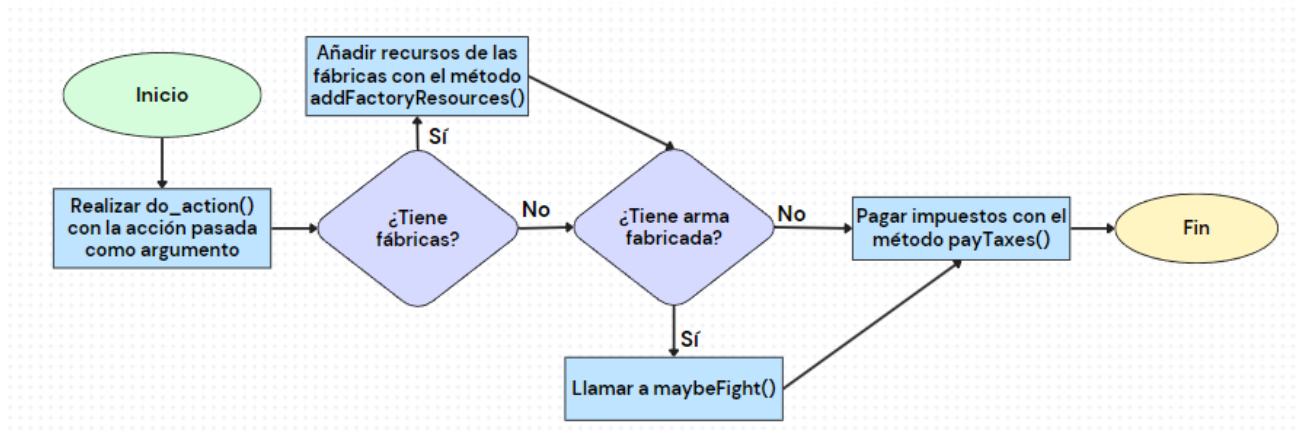


Ilustración 21. Diagrama de flujo de step() de la clase Player

En esta última iteración también se llevaron a cabo ciertos experimentos con distintos comportamientos generados con mutaciones y comportamientos diferentes a los preestablecidos, pero debido a su extensión serán comentados en mayor profundidad en el capítulo siguiente.





## Capítulo 6. Pruebas y evaluación del sistema

Este capítulo contiene el análisis y estudio intensivo del sistema completo. Además, como se ha comentado en el capítulo anterior, se estudian ciertas mutaciones y nuevos comportamientos introducidos de manera dinámica. Estas pruebas se realizan con el objetivo de justificar la capacidad del sistema para introducir en ejecución cambios en sus agentes, así como observar cómo este alcanza un nuevo equilibrio en función de los diferentes cambios aplicados.

Con el sistema completo explicado, se proceden a realizar ejecuciones para comprobar que todas las implementaciones funcionan de manera deseada, estudiando los tiempos y los comportamientos que resultan ganadores en las mismas. Aunque los ganadores no son relevantes, permiten obtener una idea de las conductas que mejor desempeñan en el sistema. Este sistema se puede ejecutar de dos formas en función de las necesidades:

- Si se quiere visualizar la interfaz gráfica, se debe ejecutar el archivo “**main.py**”.
- Si se quieren realizar ejecuciones largas sin interfaz gráfica, se debe ejecutar el archivo “**model.py**” y modificar sus parámetros de ejecución en el método **run\_model()**, para poder ajustar el número de simulaciones que se desean realizar. Todas estas ejecuciones se guardan en la carpeta “**saves**”, con su respectivo número de semilla para permitir la reproducción de esa misma simulación en la interfaz gráfica.

Las pruebas se dividen en las dos formas de ejecución. Primero se ejecuta el sistema desde la interfaz gráfica para observar que todos los elementos funcionan correctamente. Por último, se realizan 50 ejecuciones largas en las que se decidirá como ganador al primer agente que consiga obtener un total de 25.000 puntos estelares. Si el sistema alcanza los 100.000 turnos simulados, el ganador será el comportamiento con más presencia, en caso de empate, el ganador será el comportamiento que acumule la mayor suma de puntos estelares en total. Este número de turnos es el máximo que permite ejecutar la interfaz gráfica de Mesa, así si se desea probar con la interfaz se podrá visualizar completamente.

Gracias a la interfaz gráfica y a los mensajes escritos por terminal, se puede observar cómo el sistema se desarrolla de manera correcta, siguiendo fielmente la lista de prioridades de cada uno y ajustándose a los cambios de comportamiento especificados en el método **changeBehaviour()**, descrito en el capítulo anterior por medio de la Tabla 4. En la Ilustración 22 se muestra un

ejemplo de este tipo de mensajes, donde para cada agente se especifica su “id” seguido de la acción seleccionada y su número correspondiente en ACTION\_SPACE. En el caso específico de “Move”, se indica la dirección del movimiento y, si corresponde, la posición del agente al que se quiere perseguir o escapar. En caso de no tener un agente objetivo, la lista aparecerá vacía.

```
{ "type": "get_step", "step": 5 }
agent: 1 elige la acción (('Move', 'To_Planet', []), 7)
agent: 2 elige la acción (('Move', 'To_Player', []), 0)
agent: 3 elige la acción (('Move', 'To_Planet', [(4, 9)]), 3)
{ "type": "get_step", "step": 6 }
agent: 1 elige la acción ('Factory', 8)
agent: 2 elige la acción (('Move', 'To_Player', []), 3)
agent: 3 elige la acción ('Factory', 8)
{ "type": "get_step", "step": 7 }
agent: 1 elige la acción ('Factory', 8)
agent: 2 elige la acción (('Move', 'To_Player', []), 3)
agent: 3 elige la acción ('Weapon', 10)
```

Ilustración 22. Mensajes escritos por terminal

Las 50 ejecuciones obtienen como ganador en todas ellas al comportamiento Explorer, en la Tabla 5 se puede observar los resultados de turnos y tiempos, en ella se contempla cómo la media y la mediana de tiempos es considerablemente diferente, esto se debe a la existencia de algunas ejecuciones anómalas que se alargan más de lo que deberían. Destacando una ejecución que alcanzó el máximo de turnos permitidos en 316 segundos, con una abundancia de Chaser representando el 58% de agentes del sistema, seguido de una presencia menor de Explorer y Farmer (ambos con un 21% de aparición).

Si se comparan estos porcentajes de aparición con los obtenidos al estudiar las 50 simulaciones se observan datos similares. **Explorer** cuenta con un **27%**, **Chaser** con la mayor presencia con **54%** y, por último, **Farmer** con un **19%**. Gracias a estos datos se puede entender que se necesitan modificar ciertas conductas para obtener un sistema con mayor balance en la aparición de los agentes. Estas simulaciones finalizan con un total de 716 agentes, contando con una media de 14 agentes por ejecución.

Tabla 5. Resultados de sistema completo

Turnos medios	Mediana de turnos	Tiempo medio	Mediana de tiempos
13.243	9.094	32 s	19 s

## 6.1. Cambio dinámico en la estructura

Este apartado tiene como objetivo introducir modificaciones en los diferentes comportamientos preestablecidos mientras el programa se encuentra en ejecución, para posteriormente observar de qué manera afectan estos cambios, justificando así la introducción dinámica de código y la emergencia de distintas formas de actuar ante las mismas situaciones.

Gracias a las semillas obtenidas en las 50 simulaciones anteriores, se pueden recuperar y estudiar de manera visual las ejecuciones que presentan mayor interés para el experimento específico. Este proceso para recuperar ejecuciones se realiza al iniciar la interfaz gráfica, donde el sistema pregunta si se desea utilizar una semilla personalizada, en caso positivo se debe introducir el valor deseado y se establece por medio del método **random.seed()**, en caso negativo la semilla será determinada aleatoriamente con un valor entre 0 y 10.000.

### 6.1.1. Nueva clase con cambio de conducta

Para esta prueba se toman en cuenta las ejecuciones que tienen como ganador al agente con id 1, correspondiente al comportamiento Explorer, ya que estos agentes consiguen sobrevivir en el sistema desde el principio sin ser eliminados. En las 50 simulaciones, este suceso ocurre en 9 de ellas.

Para poder ilustrar las modificaciones se ejecuta en profundidad la semilla 1847, utilizada en la ejecución número 14 de la carpeta “saves/Normal”. El cambio dinámico ocurre en el turno 20 de la simulación (Ilustración 23), en este momento, la primera instancia del comportamiento Explorer es modificado por una nueva clase generada de manera dinámica llamada **DummyExplorer**, está hereda de Explorer, para que cumpla la condición de ser un comportamiento. Además, se añade un nuevo método de manera dinámica a esta clase para sobrescribir el método polimórfico **changeBehaviour()**, modificando su funcionamiento eliminando la habilidad de evitar luchas y construir fábricas.

La creación de la clase en tiempo de ejecución se realiza gracias a la función predefinida en Python “**type()**”. En cambio, la asignación específica de un método se realiza directamente para toda la clase, reemplazando el método original por el nuevo mediante una asignación.

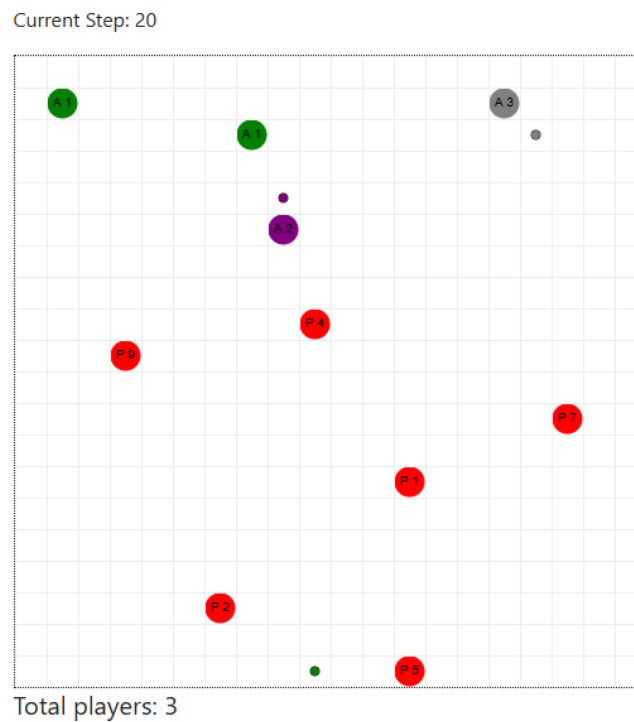


Ilustración 23. Estado del sistema al producir cambio en primer Explorer

Tabla 6. Evolución del sistema tras la modificación en primer Explorer

**Evolución del sistema tras cambio en agente 1 (verde)**

**Current Step: 48**

Total players: 3

**Ilustración 24.** Chaser (morado) encuentra a agente 1

**Current Step: 115**

Total players: 4

**Ilustración 25.** Chaser conquista planetas de agente 1

La Tabla 6 presenta 2 ilustraciones, que representan la evolución del sistema tras la modificación. Al producir el cambio, los agentes siguen actuando de manera normal, esto se debe a que el jugador modificado se encuentra muy alejado de los otros agentes, conquistando todos los planetas que ve a su alrededor. Pero debido a la **falta de “inteligencia” para evitar luchas**, en la Ilustración 24 (representación del turno 48), se observa cómo es atacado por el agente morado (Chaser) que comienza a perseguirle y luchar contra él. Por tanto, cuando se llega al turno 115, representado por la Ilustración 25, se observa cómo el Chaser ha conquistado la mayoría de los planetas del sistema y sigue en persecución del agente modificado. En estos 67 turnos de diferencia el agente morado consigue aumentar sus victorias en batallas desde las 4 del turno 48 a las 42 del turno 115. Posteriormente, el agente modificado es eliminado en el turno 200, por no conseguir un balance positivo en este ciclo de 100 turnos. La Ilustración 26 representa la leyenda del último turno de este agente en la simulación, las diferentes leyendas de los turnos estudiados en este apartado se encuentran presentes en el Anexo C.

■ Agent: 1 Resources: T: 621 G: 988 P: 2 F: 11 Stellar Points: 222 Balance: -22  
Behaviour: DummyExplorer Weapon: N Battles Won: 0 Upgrades: None

■ Agent: 2 Resources: T: 7451 G: 6658 P: 0 F: 7 Stellar Points: 665 Balance: 516  
Behaviour: Chaser Weapon: Guided missil Battles Won: 100 Upgrades: D ↑

■ Agent: 3 Resources: T: 2536 G: 2691 P: 0 F: 28 Stellar Points: 141 Balance: 65  
Behaviour: Farmer Weapon: Guided missil Battles Won: 17 Upgrades: F ↑

■ Agent: 4 Resources: T: 4552 G: 6366 P: 0 F: 7 Stellar Points: 34 Balance: 34  
Behaviour: Chaser Weapon: Guided missil Battles Won: 26 Upgrades: D ↑

Ilustración 26. Leyenda del último turno de DummyExplorer en la ejecución

Tras analizar este ejemplo concreto, se decide simular las 50 ejecuciones probadas en el apartado anterior para determinar la diferencia que ocasiona esta modificación dinámica.

Tabla 7. Resultados con cambio dinámico en primer Explorer

Media agentes	Presencia Explorer	Presencia Chaser	Presencia Farmer
14	26 %	54 %	20 %
Turnos medios	Mediana de turnos	Tiempo medio	Mediana de tiempos
15.836	10.688	40 s	24 s

Estas 50 simulaciones finalizan con un total de 736 agentes, lo que supone **un aumento del 2%** en comparación con los resultados por defecto, tanto la media de agentes como los porcentajes de aparición presentes en la Tabla 7 se mantienen iguales a excepción de la disminución de Explorer y el aumento de Farmer (1% en ambos casos). En cuanto a los turnos, representan un aumento tanto en la media (20%) como en la mediana (17%), por consecuencia los tiempos medidos aumentan respecto a los obtenidos en la Tabla 5. Este aumento se debe a que, en 9 de las anteriores simulaciones el agente con id 1 era el ganador, pero al modificarlo la ejecución se alarga obteniendo nuevos Explorer como vencedores.

Al observar estos resultados se decide volver a ejecutar las 50 simulaciones, pero en esta ocasión se modifica la funcionalidad para toda la clase Explorer, así los nuevos agentes que surjan en el sistema tendrán la imposibilidad de huir de las luchas.

Tabla 8. Resultados con cambio dinámico en todos los Explorer

Media agentes	Presencia Explorer	Presencia Chaser	Presencia Farmer
17	23 %	57 %	20 %
Turnos medios	Mediana de turnos	Tiempo medio	Mediana de tiempos
87.413	100.000	243 s	270 s

En esta ocasión las simulaciones acaban con un total de 861 agentes, lo que supone **un aumento del 20%** en comparación con las simulaciones originales. Además, el cambio en todos los Explorer se ve reflejado en los porcentajes de aparición de la Tabla 8, reduciendo la presencia de Explorer en un 3% y aumentando la de Chaser (3%) si se compara con la Tabla 7. Por otro lado, al estudiar los turnos se observan datos que en la mediana se corresponden con el límite establecido, lo que se traduce en que ningún agente ha conseguido obtener los puntos estelares necesarios para ganar a excepción de Explorer, que aún sin la capacidad de huir, lo consigue en 9 ocasiones.

Tanto las ejecuciones cambiando dinámicamente la primera instancia, como las ejecuciones en las que se cambia a nivel global se encuentran en la carpeta “save/CambioExplorer” presente en el Anexo B.

### 6.1.2. Nuevo atributo utilizado en el comportamiento

La modificación llevada a cabo en este comportamiento se centra en modificar la primera instancia de la clase Chaser (agente con id 2) permitiéndole identificar al objetivo con mayor puntuación estelar y seguirle por todo el tablero. Con el objetivo de reducir su puntuación a través de la lucha, equilibrando el sistema y permitiendo a los demás agentes sobrevivir.

La implementación desarrollada en este experimento se centra en la creación de una nueva clase llamada **Agressive**, que únicamente hereda de Chaser y no presenta un comportamiento diferente por sí misma. Cuando la ejecución alcance el turno 50, se añade **un nuevo atributo** a la clase por medio de **setattr()**, llamado “agent\_more\_points”, que será utilizado por un nuevo método **changeBehaviour()** asignado a la clase en ejecución, incluyendo así la nueva funcionalidad. La Ilustración 27 representa el diagrama que explica en profundidad la implementación de este nuevo método.

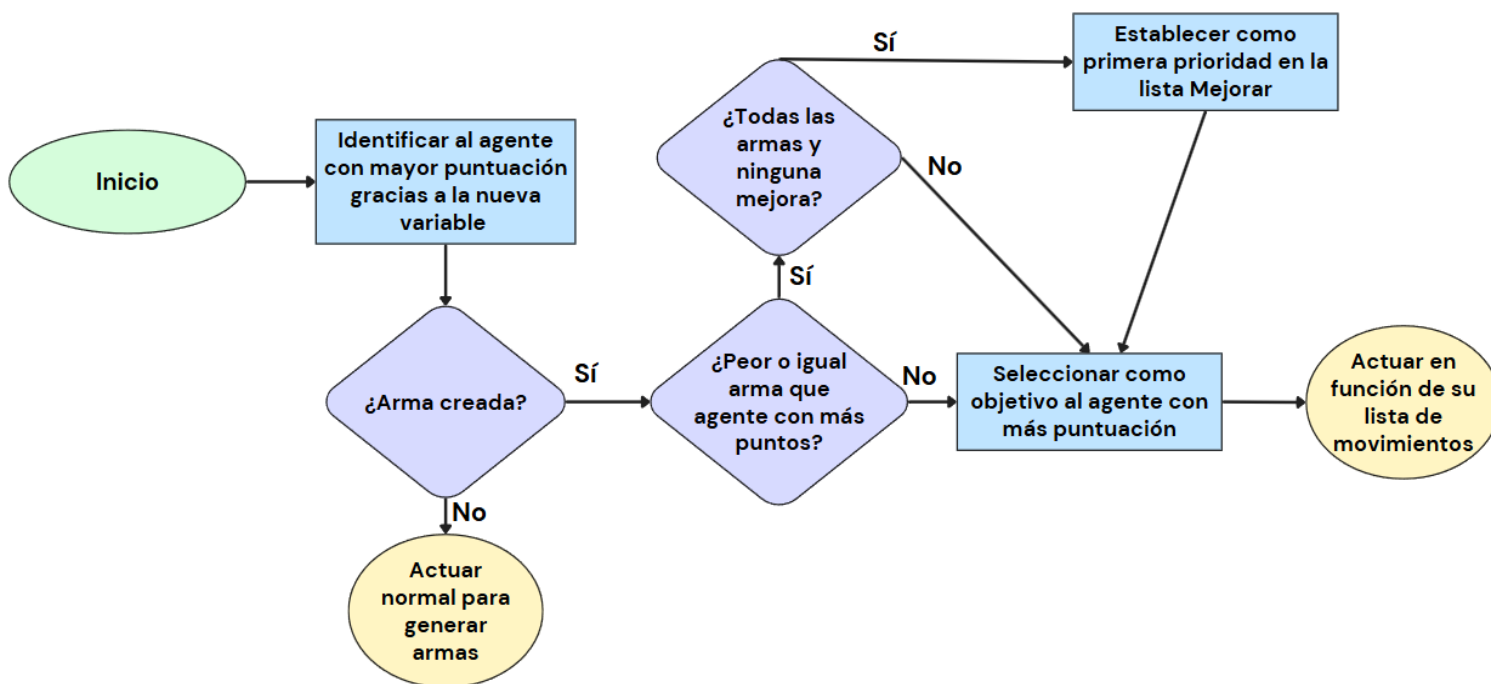


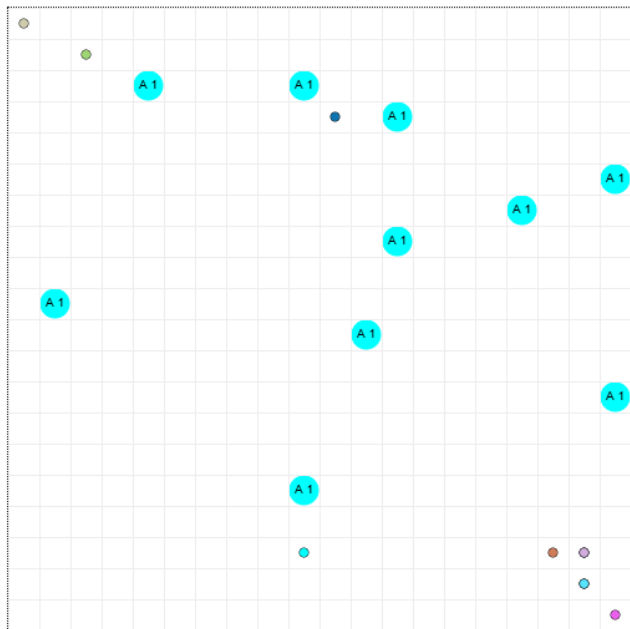
Ilustración 27. Diagrama changeBehaviour Agressive

Para ejemplificar estos cambios dinámicos se hace uso de la semilla 1390, presente en la ejecución número 2 de “saves/Normal” y cuyo ganador es el agente con id 1 (Explorer). En las pruebas se ejecuta el sistema dos veces, en la primera de ellas no se realiza una modificación del comportamiento, con el objetivo de poder identificar el turno en el que el agente a estudiar es eliminado del sistema y en la última ejecución se produce la modificación en el turno 50.

Tabla 9. Comparación de ejecuciones en el último turno de agente 2 (Chaser)

### Ejecución sin modificación dinámica

Current Step: 1399



Total players: 10

Ilustración 28. Último turno de agente 2

■ Agent: 1 Resources: T: 18762 G: 2481 P: 10 F: 40 **Stellar Points: 5383** Balance: 722  
Behaviour: **Explorer** Weapon: **Guided missil** Battles Won: 124 Upgrades: **None**

■ Agent: 2 Resources: T: 13804 G: 10352 P: 0 F: 38 **Stellar Points: 566** Balance: -7  
Behaviour: **Chaser** Weapon: **Guided missil** Battles Won: 972 Upgrades: **D ↑**

■ Agent: 8 Resources: T: 18439 G: 15411 P: 0 F: 39 **Stellar Points: 399** Balance: 65  
Behaviour: **Chaser** Weapon: **Guided missil** Battles Won: 697 Upgrades: **D ↑**

■ Agent: 10 Resources: T: 14846 G: 10486 P: 0 F: 38 **Stellar Points: 555** Balance: 2  
Behaviour: **Farmer** Weapon: **Guided missil** Battles Won: 143 Upgrades: **F ↑**

■ Agent: 11 Resources: T: 4905 G: 6347 P: 0 F: 32 **Stellar Points: 227** Balance: -18  
Behaviour: **Explorer** Weapon: **Plasma Cannon** Battles Won: 64 Upgrades: **None**

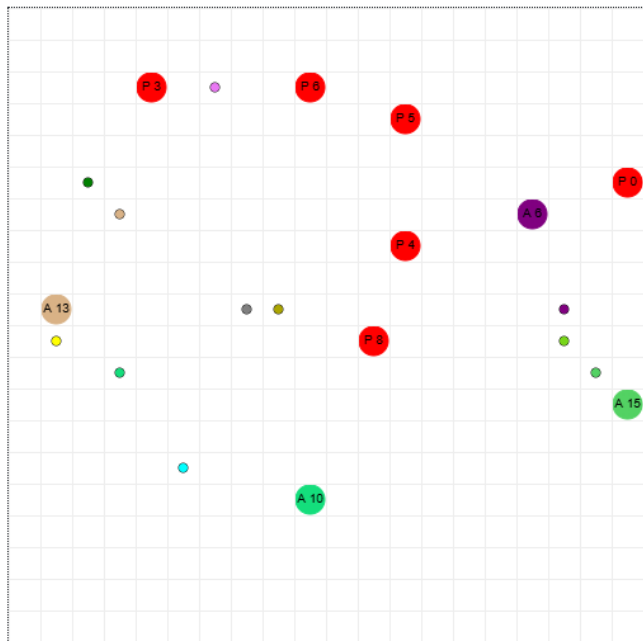
■ Agent: 12 Resources: T: 12111 G: 14250 P: 0 F: 37 **Stellar Points: 38** Balance: -9  
Behaviour: **Farmer** Weapon: **Guided missil** Battles Won: 217 Upgrades: **F ↑**

■ Agent: 13 Resources: T: 18200 G: 14621 P: 0 F: 38 **Stellar Points: 99** Balance: 17  
Behaviour: **Chaser** Weapon: **Guided missil** Battles Won: 290 Upgrades: **D ↑**

Ilustración 29. Leyenda simplificada de Ilustración 28

### Ejecución con modificación dinámica en turno 50

Current Step: 1399



Total players: 11

Ilustración 30. Estado con la modificación dinámica

■ Agent: 1 Resources: T: 14383 G: 24947 P: 0 F: 39 **Stellar Points: 1506** Balance: 29  
Behaviour: **Explorer** Weapon: **Guided missil** Battles Won: 199 Upgrades: **None**

■ Agent: 2 Resources: T: 28175 G: 27025 P: 0 F: 13 **Stellar Points: 2090** Balance: 179  
Behaviour: **Agressive** Weapon: **Guided missil** Battles Won: 572 Upgrades: **D ↑**

■ Agent: 3 Resources: T: 16396 G: 27601 P: 0 F: 41 **Stellar Points: 1140** Balance: 25  
Behaviour: **Farmer** Weapon: **Guided missil** Battles Won: 467 Upgrades: **F ↑**

■ Agent: 5 Resources: T: 16322 G: 15305 P: 0 F: 40 **Stellar Points: 778** Balance: 45  
Behaviour: **Farmer** Weapon: **Guided missil** Battles Won: 364 Upgrades: **F ↑**

■ Agent: 6 Resources: T: 31280 G: 41563 P: 1 F: 42 **Stellar Points: 755** Balance: 63  
Behaviour: **Chaser** Weapon: **Guided missil** Battles Won: 839 Upgrades: **D ↑**

■ Agent: 7 Resources: T: 5030 G: 9234 P: 0 F: 37 **Stellar Points: 904** Balance: -25  
Behaviour: **Explorer** Weapon: **Plasma Cannon** Battles Won: 163 Upgrades: **None**

■ Agent: 9 Resources: T: 27528 G: 32459 P: 0 F: 42 **Stellar Points: 474** Balance: 46  
Behaviour: **Chaser** Weapon: **Guided missil** Battles Won: 567 Upgrades: **D ↑**

Ilustración 31. Leyenda simplificada de Ilustración 30



La Tabla 9 agrupa los estados de las dos ejecuciones en el turno 1.399, este en la ejecución normal (Ilustración 28) representa el último turno del agente con id 2, aquí el agente con id 1 presenta un **dominio total de todos los planetas del tablero**. Además, en la Ilustración 29, se detalla la leyenda simplificada para este punto de la simulación, observándose la gran diferencia de puntos estelares que presenta el jugador 1 en comparación a sus rivales, con un balance positivo de 722 puntos en este ciclo de 100 turnos. Gracias a esta leyenda, se muestra cómo el agente eliminado en el siguiente turno será el número 2, al ser el jugador con balance negativo más antiguo con un total de -7.

Al comparar estos datos con la ejecución modificada en el turno 50, los resultados son totalmente contrarios. En la Ilustración 30, se observa cómo en esta ejecución **no existe un dominio de planetas conquistados**, sino que se encuentran en disputa constante. La Ilustración 31, presenta la leyenda en la que se puede ver una igualdad de puntuaciones más equilibrada que la Ilustración 29. El agente ganador en este caso es el modificado y su objetivo a perseguir es el agente con id 1, al ser el siguiente con más puntos estelares. Por este motivo, se presenta la diferencia del sistema, ya que en la anterior ejecución el jugador 2, se centra en perseguir al agente con peor arma en su entorno o en toda la simulación, sin tener en cuenta al agente 1, que huye y conquista todos los planetas.

Por tanto, el objetivo propuesto para este cambio en ejecución se cumple, consiguiendo alterar el estado del sistema y adquiriendo un cierto nivel de equilibrio en el que ningún agente destaca por encima de otros (véase Ilustración 31), alargando la ejecución de 5.705 a 13.234 turnos. El agente modificado consigue sobrevivir hasta el turno 4299 (leyenda presente en el Anexo C) y dado que solo existe este comportamiento agresivo, cuando es eliminado, la ejecución se desarrolla de manera normal, siendo el ganador un Explorer.

Tras analizar este ejemplo concreto, se ejecutan las 50 semillas anteriores con el objetivo de comparar los resultados obtenidos con las ejecuciones normales. El número total de agentes que finalizan las 50 ejecuciones es de 823, **un 15% de aumento respecto a los obtenidos con el sistema sin cambios**.

Tabla 10. Resultados con cambio dinámico en primer Chaser

Media agentes	Presencia Explorer	Presencia Chaser	Presencia Farmer
16	26 %	53 %	21 %
Turnos medios	Mediana de turnos	Tiempo medio	Mediana de tiempos
29.540	17.933	90 s	48 s

En la Tabla 10, se observa cómo los porcentajes de presencia se mantienen similares, debido a que solo se modifica la primera instancia de Chaser, por lo que cuando es eliminado el sistema se desarrolla de manera normal. Sin embargo, este cambio en la primera instancia supone un aumento que representa más del doble en los turnos medios (123%) y el 97% de aumento en la mediana de turnos, lo que se traduce en, un aumento de los tiempos medidos, verificando los datos obtenidos en el ejemplo ilustrado anteriormente.

Al ver los buenos resultados de este experimento, se decide comprobar que pasaría si el cambio se realiza a nivel global para todos los agentes Chaser.

Tabla 11. Resultados con cambio dinámico en todos los Chaser

Media agentes	Presencia Explorer	Presencia Chaser	Presencia Farmer
23	32 %	29 %	39 %
Turnos medios	Mediana de turnos	Tiempo medio	Mediana de tiempos
100.000	100.000	375 s	369 s

En esta ocasión las simulaciones acaban con **1.171 agentes**, suponiendo **un incremento del 64%** en comparación a las simulaciones por defecto y un 42% si se compara cuando solo se cambia la primera instancia. Además, en la Tabla 11 se observa cómo la presencia de Chaser se reduce en un 25% con los datos obtenidos en las simulaciones sin cambios, esto ocurre porque estos agentes modificados al ser tan competitivos y perseguir al agente ganador, siempre coinciden en la dirección de movimiento, por lo que se acaban destruyendo entre ellos. Dando lugar a comportamientos como Farmer que pasa de ser el comportamiento con menor presencia al más abundante.

En cuanto a los turnos, estos siempre alcanzan el límite en las 50 semillas probadas, por lo que los ganadores acaban siendo los agentes más abundante. Tanto las pruebas recogidas en la Tabla 10 como en la Tabla 11 se encuentran presentes en la carpeta “*saves/CambioChaser*” en el Anexo B.

### 6.1.3. Nuevo método y modificación de atributos

En este apartado se introduce dinámicamente una nueva conducta que busca mejorar la supervivencia del comportamiento Farmer en el sistema, debido a que, en las pruebas con el sistema por defecto no consigue tener una aparición consistente y, por lo tanto, no gana en ninguna de las simulaciones.

Para probar este nuevo comportamiento emergente se decide modificar en el turno 10 el agente con id 3 (Farmer), creando una nueva clase dinámica llamada *OptimalFarmer*, esta creación se realiza de la misma manera que en el apartado 6.1.1, con la pequeña diferencia de que esta vez la clase hereda del comportamiento Farmer.

Una vez asignada la nueva clase al agente, dinámicamente se crea un nuevo método llamado **check\_money()**, cuya funcionalidad se basa en comprobar si el agente puede crear una nueva fábrica y seguir teniendo dinero para mantener sus planetas conquistados, dotando a este agente de más inteligencia para gestionar sus recursos. Además, se modifica en su diccionario de acciones la habilidad de obtener una mejora de daño a través del método **resetBehaviour()**, encargado de reestablecer el comportamiento para actuar de manera predeterminada (reiniciando su lista de prioridades y diccionario de acciones).

Estos nuevos cambios se añaden en un nuevo método que sobrescribirá al método *changeBehaviour()*, encargado de decidir en ejecución la acción prioritaria. Los principales cambios de este método respecto al método original de la clase Farmer, se basan en las diferencias de actuar si solo tiene jugadores adyacentes y si no tiene nada alrededor.

- **Si solo tiene jugadores en casillas adyacentes.** Comprueba si tiene armas creadas.
  - En caso de tener arma. Verifica si tiene mejor arma que sus enemigos.

- Si tiene mejor arma. Persigue al que peor arma tenga, priorizando el movimiento.
- Si no la tiene. Comprueba si su arma es peor o si el otro agente tiene mejora de daño activa.
  - Si es el caso, huye de ese agente.
  - Si no es el caso, comprueba si tiene planetas adyacentes y si no los tiene, actúa como si no hubiera nada en su entorno.
- En caso de no tener arma, huye, priorizando la acción de moverse.
- **Si no tiene nada en casillas adyacentes.** Verifica si puede crear fábricas manteniendo sus planetas conquistados utilizando el método `check_money()`.
  - En caso positivo, actúa normal.
  - En caso negativo, modifica su dirección de movimiento para moverse hacia los planetas, así si la única acción posible es el movimiento se dirigirá hacia el planeta más cercano.

En esencia, estas nuevas modificaciones dotan al comportamiento `OptimalFarmer` de la capacidad de evitar luchas con agentes mejor armados y le permiten la habilidad de gestionar sus recursos, maximizando los puntos obtenidos por sus planetas conquistados. Debido a que, si el agente gasta todos sus recursos en construir fábricas, no podrá asumir el coste que suponen los impuestos de los planetas, perdiendo su principal fuente de puntos estelares.

A continuación, se muestran las leyendas del agente con id 3 en dos ejecuciones de la misma semilla 4472, esta se corresponde a la ejecución número 44 de la carpeta “`saves/Normal`”.

■ **Agent: 3** Resources: T: 1672 G: 3546 P: 0 F: 26 **Stellar Points: 77** Balance: -9  
 Behaviour: **Farmer** Weapon: **Guided missil** Battles Won: 25 Upgrades: F ↑

Ilustración 32. Leyenda del turno 199 del agente sin modificar

■ **Agent: 3** Resources: T: 4338 G: 3306 P: 4 F: 27 **Stellar Points: 472** Balance: 318  
 Behaviour: **OptimalFarmer** Weapon: **Guided missil** Battles Won: 47 Upgrades: D ↑ F ↑

Ilustración 33. Leyenda del turno 199 del agente modificado

La Ilustración 32, representa los datos referentes al último turno (199) en el que el agente Farmer sobrevive en esta semilla. Por otro lado, en la Ilustración 33, se muestran los datos de este agente, pero con la diferencia que esta vez se ha modificado dinámicamente (en el turno 10) para simular la aparición del comportamiento explicado al principio del apartado. Se puede observar la diferencia de puntos estelares, pero destaca considerablemente la diferencia de balances, en la Ilustración 32 el agente está con balance negativo, por no poder evitar luchas ni gestionar correctamente sus recursos, mientras que en la Ilustración 33 es el jugador con una ganancia mayor en este ciclo de 100 turnos. Cabe destacar, que el agente modificado resulta ganador en esta simulación, consiguiendo llegar a los 25.000 puntos estelares en 11.892 turnos. En el Anexo C se encuentran las leyendas completas, así como los estados de la simulación en este turno.

Tras probar la modificación en esta semilla concreta, se decide comprobar los resultados que se obtienen al ejecutar las 50 simulaciones anteriores. La Tabla 12, presenta los resultados obtenidos aplicando la modificación dinámica en el turno 10.

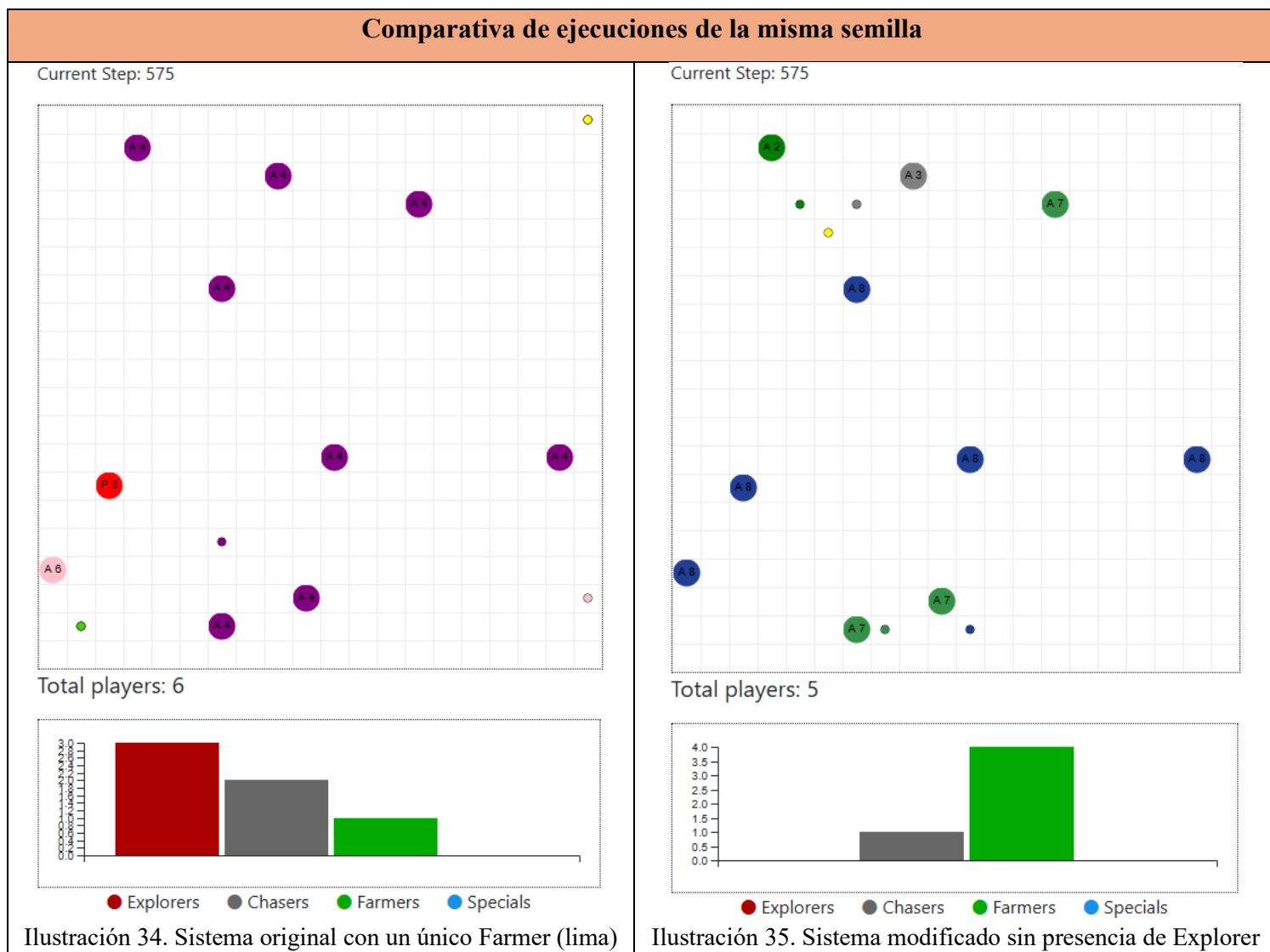
Tabla 12. Resultados con cambio dinámico en primer Farmer

Media agentes	Presencia Explorer	Presencia Chaser	Presencia Farmer
16	21 %	53 %	26 %
Turnos medios	Mediana de turnos	Tiempo medio	Mediana de tiempos
16.855	10.928	46 s	25 s

Con esta modificación de la primera instancia de Farmer las simulaciones finalizan con un total de 794 agentes, lo que supone **un incremento del 10%** en comparación con las simulaciones sin cambios. En cuanto a los porcentajes de la Tabla 12, muestran cómo Explorer ha reducido su aparición en un 7%, aumentando la presencia de Farmer, esto se debe a que el agente modificado logra sobrevivir en 33 de las 50 simulaciones, cuando en las ejecuciones originales no sobrevive en ninguna de ellas. Por otro lado, los turnos estudiados representan un aumento considerable tanto en la media (27%) como en la mediana (20%) en comparación a las pruebas sin modificación. La instancia de Farmer modificada logra ganar en 24 simulaciones, demostrando que el cambio dinámico es efectivo.

Como en los apartados anteriores se vuelve a ejecutar las 50 semillas, pero con la diferencia que esta vez se modifica la clase a nivel global. Con el objetivo de visualizar el impacto que esta mutación tiene en el sistema, se ilustra mediante la Tabla 13, la cual contiene dos imágenes comparativas del estado en el mismo turno (575) de la semilla 4472.

Tabla 13. Comparación entre sistema original y sistema con modificación global en Farmer



La Ilustración 34, hace referencia a la ejecución normal del sistema y en ella se observa un domino de planetas por parte del agente morado (Explorer), principalmente destaca que en este instante la ejecución solo presenta un agente con comportamiento Farmer, demostrando que es el comportamiento que peor se adapta al sistema original. Sin embargo, la Ilustración 35, presenta el mismo turno, pero con la clase Farmer modificada para adecuarse a los cambios

comentados anteriormente, aquí se observa cómo la modificación implementada a nivel global en la clase permite obtener un nuevo balance en el sistema, donde las presencias de comportamientos se han visto revertidas, pasando a ser Farmer el comportamiento con más aparición y Explorer el que peor se está adaptando. A pesar de esto, la simulación finaliza con un agente Explorer como ganador en el turno 23.041.

La Tabla 14, representa los resultados globales obtenidos al ejecutar las 50 semillas con la modificación global en la clase Farmer.

Tabla 14. Resultados con cambio dinámico en todos los Farmer

Media agentes	Presencia Explorer	Presencia Chaser	Presencia Farmer
22	18 %	31 %	51 %
Turnos medios	Mediana de turnos	Tiempo medio	Mediana de tiempos
44.431	30.503	169 s	106 s

Estas 50 simulaciones acaban con **1.106 agentes**, lo que supone **un incremento del 54%** en comparación con los resultados por defecto y **un 39%** con los sistemas que **solo modifican la primera instancia**. Destaca principalmente, el aumento de presencia de Farmer, desbancando a Chaser como el más abundante en el sistema y reduciendo la presencia de Explorer un 11% con el sistema original. Además, debido al aumento de competitividad en el sistema, los turnos medidos y en su defecto los tiempos se ven incrementados, destacando el aumento de tiempo medio de 267% en comparación con los resultados de la Tabla 12. Este gran aumento se debe, en parte, a la cantidad de nuevas comprobaciones que deben realizar los agentes Farmer.

En cuanto a los vencedores en las simulaciones, al aplicar este cambio a nivel global los únicos agentes que consiguen la victoria antes de los turnos límites son los Explorer, lo que supone un retroceso en cuanto a efectividad con el sistema que solo cambia la primera instancia, ya que en este sistema Farmer consigue ganar en 24 ocasiones. En las 11 ejecuciones donde se alcanzan el límite de turnos Farmer se lleva la victoria en 9 ocasiones y Chaser solo lo consigue 2 ejecuciones.

Tanto las pruebas recogidas en la Tabla 12 como en la Tabla 14 se encuentran presentes en la carpeta “*saves/CambioFarmer*” en el Anexo B.

En estos 3 subapartados, se ha demostrado la capacidad del sistema para soportar cambios emergentes y cómo estos nuevos comportamientos balancean el sistema independientemente de si son ventajosos (**OptimalFarmer**) o perjudiciales (**DummyExplorer**), de manera similar a lo que ocurriría en un modelo del mundo real. A continuación, se presenta un resumen con los 3 cambios dinámicos realizados en este apartado:

- **Cambio perjudicial (**DummyExplorer**).** Con los cambios introducidos en ejecución se puede entender el factor diferencial que supone la capacidad de evitar las luchas por parte de Explorer, ya que en la misma ejecución este pequeño cambio determina la supervivencia del agente o su eliminación. Por otro lado, el agente Chaser se ve considerablemente beneficiado al poder enfrentarse a un agente que no huye y que no tiene ningún arma para defenderse.
- **Cambio neutro (**Agressive**).** El cambio parece beneficioso por la forma en la que permite al agente sobrevivir unos turnos más en la simulación, pero al aplicarlo a todos los agentes Chaser se observan sus puntos negativos, llegando a acabar con la presencia dominante de este agente.
- **Cambio beneficioso (**OptimalFarmer**).** Tanto el cambio a nivel individual, como el cambio a todos los agentes Farmer es claramente beneficioso, permitiéndole ganar ejecuciones y permanecer como el más abundante cuando el cambio es a nivel global. Esto se debe a la introducción de una mayor inteligencia para gestionar sus recursos y la incorporación de la capacidad de huir, característica propia de los agentes Explorer.

## 6.2. Nuevos comportamientos

Al contrario que en los apartados anteriores, en este se prueban nuevas modificaciones observando cómo se modifica el estado del sistema al introducir comportamientos radicalmente diferentes. Las modificaciones que se estudiarán serán las siguientes: Agentes con comportamientos aleatorios (**RandomBehaviour**), agentes con un comportamiento considerado óptimo (**CustomBehaviour**) y agentes que comparten sus recursos y huyen de la lucha (**Friendly**).



La realización de las pruebas se efectuará de la misma manera, ejecutando las 50 semillas del apartado 6.1, donde cada 100 turnos se introducirá un nuevo agente. Aunque, dependiendo de la modificación se realizarán cambios específicos para estudiar y comparar la evolución del sistema al introducir estos nuevos comportamientos y demostrar la versatilidad de cambios que permite.

Todas las pruebas están presentes en el Anexo B en la carpeta “saves”, con el objetivo de entender los resultados presentes en estas carpetas, se explica brevemente su composición. La información de los jugadores viene dividida en listas, la primera lista contiene la información referente a su posición, recursos, planetas conquistados, fábricas generadas, arma y por último sus puntos estelares. Los siguientes valores serán las batallas ganadas, las mejoras creadas y listas con la información referente a su comportamiento (lista de prioridad, dirección de movimiento y mejoras disponibles).

### 6.2.1. Agentes con comportamientos aleatorios (**RandomBehaviour**)

Como se comentó en la explicación de las clases Farmer, Explorer y Chaser, para poder añadir un nuevo comportamiento se debe generar una clase (**RandomBehaviour**) que herede de la clase Behaviour, en ella se modifica su lista de prioridad, su diccionario de acciones y se añaden dos variables auxiliares que copian estas dos modificaciones. Además, si se quiere cambiar su comportamiento dinámico también se deben modificar los métodos changeBehaviour() y resetBehaviour(). Para este caso concreto bastará con modificar todos sus elementos menos changeBehaviour(), porque al ser aleatorio no se quiere modificar su forma de actuar con el objetivo de poder observar si surgen algunas combinaciones ganadoras que no se habían tenido en cuenta.

En el caso de la implementación de resetBehaviour() se realiza por seguridad, por si en algún momento se modifica el método changeBehaviour() y se quiere recuperar sus prioridades iniciales. La implementación del método es simple y se basa en generar copias de las variables auxiliares creadas al inicio de la clase, recuperando así la lista de prioridades y el diccionario de acciones generados al inicio.

Para la creación aleatoria de su lista de prioridades se hace uso del método propio de esta clase **setRandomPriorities()**, cuya implementación se encuentra detallada en el diagrama de flujo de la Ilustración 36. Para determinar su dirección de movimiento y las mejoras que tendrá se hace uso del método de la clase Behaviour **getRandomSpecialActions()**, cuya función es obtener de manera aleatoria un número entre 0 y 1. Si el número es 0 el booleano correspondiente será False y si por el contrario es 1 el booleano será True.

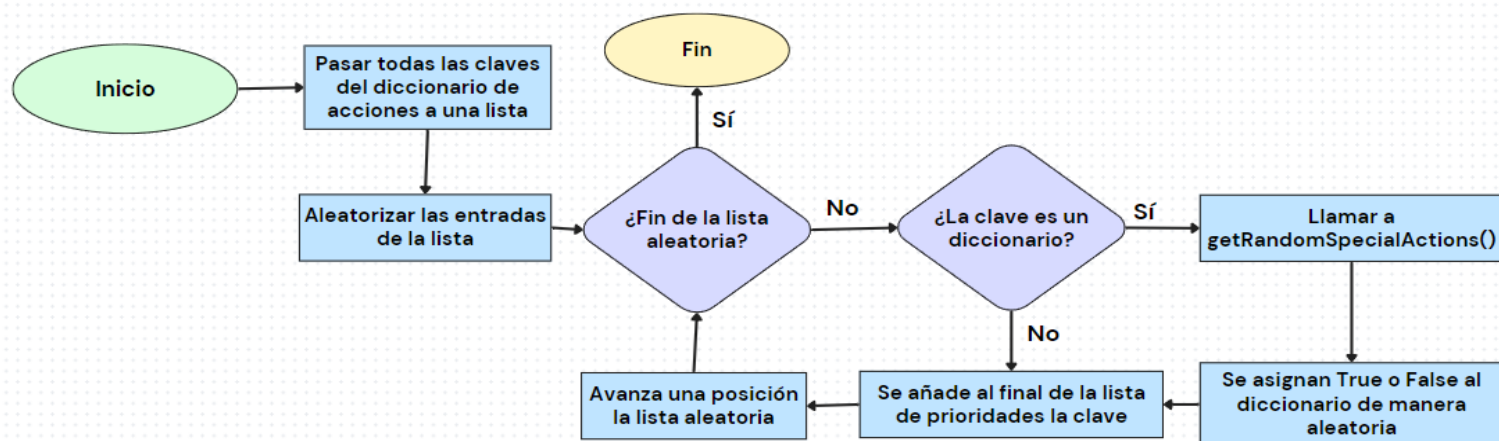


Ilustración 36. Diagrama de flujo de setRandomPriorities() de la clase RandomBehaviour

Gracias a esta implementación se obtienen agentes que se comportan de una manera aleatoria, pero para poder añadirlos al sistema, se debe modificar la forma en la que se incluyen los agentes con comportamientos, para ello se modifica el método addAgent() de la clase Game. Con esta modificación, cuando se ejecute el sistema cada 100 turnos existirá la posibilidad de obtener un agente con un comportamiento aleatorio, su nombre será Random seguido de su Id.

La Tabla 15 presenta los resultados medios obtenidos al introducir agentes con comportamientos aleatorios en las 50 simulaciones.

Tabla 15. Resultados con comportamiento RandomBehaviour

Presencia Explorer	Presencia Chaser	Presencia Farmer	Presencia Random
19 %	42 %	16 %	23 %
Turnos medios	Mediana de turnos	Tiempo medio	Mediana de tiempos
19.130	14.712	52 s	36 s

Con la introducción de este nuevo comportamiento el número final de agentes aumenta hasta los **853**, un **19% de incremento con el sistema original**. En 23 de las 50 ejecuciones el comportamiento aleatorio consigue llegar a los 25.000 puntos estelares, observando un patrón en sus listas de prioridades, donde la mayoría establece como acción menos prioritaria el movimiento. Esto se debe a que esta acción es la única del juego que solo tiene una restricción de recursos inicial, por lo que siempre que se haya efectuado este pago se podrá realizar sin problemas, ignorando los elementos que se encuentren con una prioridad menor. Así, al establecerla como última prioridad, pueden comprobar todas las demás acciones. Por otra parte, la dirección de movimiento de estos agentes siempre es hacia planetas y todos presentan al menos la mejora de daño.

Gracias a las medianas de la Tabla 15 se pueden observar cómo algunas ejecuciones anómalas alargan de manera considerable la ejecución, si se separan los tiempos por comportamientos se observa cómo en los 27 casos que Explorer es el ganador la media es de 33 segundos. Por el contrario, cuando RandomBehaviour gana, lo hace con una media de tiempo de 75 segundos. Esta diferencia de tiempo tiene sentido, debido a que Explorer siempre actúa siguiendo unas reglas establecidas. Mientras que los agentes RandomBehaviour tardan en encontrar aleatoriamente su lista de prioridad “óptima”, justificando así la diferencia entre la media y la mediana de turnos y tiempos.

### 6.2.2. Agentes con comportamiento óptimo (CustomBehaviour)

Con la idea de obtener un comportamiento que consiga permanecer el mayor ciclo de 100 turnos posible, se crea este agente personalizado (**CustomBehaviour**), que independientemente de su lista de prioridad, siempre buscará optimizar sus acciones para obtener puntos estelares. Además, gracias al análisis de los datos proporcionados por los agentes aleatorios, se puede observar que los ganadores siempre siguen una lista de prioridad similar, por lo que aprovechando este mismo esquema se generan agentes con lista de prioridades y acciones “óptimas”.

Para la implementación de esta clase se genera una nueva que hereda de la clase padre Behaviour, en ella se debe permitir la declaración de la lista de prioridades del agente, para ello

se hace uso de los métodos auxiliares **inputPriorities()** y **setPriorities()**. El diagrama de flujo del método **inputPriorities()** queda reflejado en la Ilustración 37 y es el que presenta mayor complejidad, debido al bucle infinito que comprobará que la lista sea correcta.

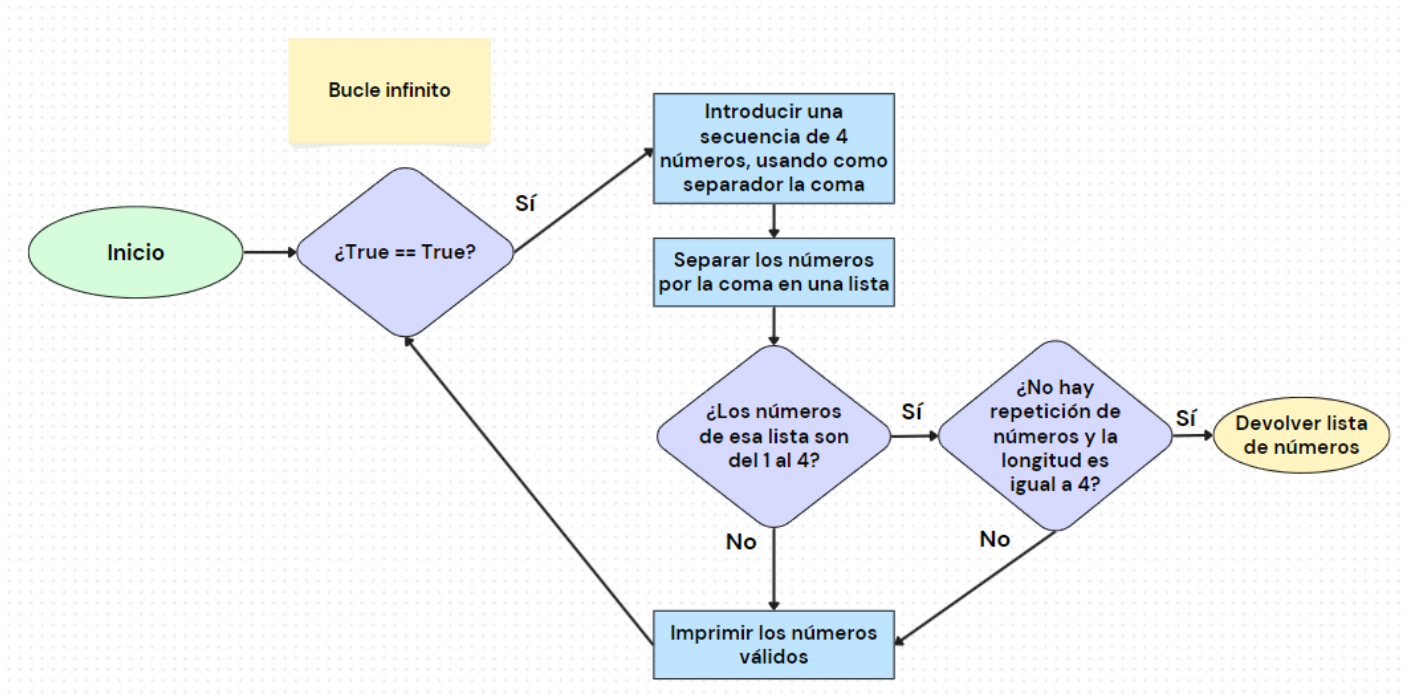


Ilustración 37. Diagrama de flujo de **inputPriorities()** de la clase **CustomBehaviour**

Por otro lado, el método **setPriorities()** en esencia es más básico y tras obtener la lista de prioridades llamando a **inputPriorities()**, pregunta la dirección de movimiento deseada a través de letras:

- **Si la letra introducida es A.** La dirección de movimiento será hacia agentes.
- **Si la letra introducida es P.** La dirección de movimiento será hacia planetas.
- **Si es cualquier otra letra.** La dirección de movimiento será aleatoria.

Para decidir las mejoras se hará de manera similar dependiendo de la letra introducida:

- **Si la letra es D.** Solo tendrá la mejora de daño disponible.
- **Si la letra es F.** Solo tendrá la mejora de recursos disponible.
- **Si la letra es B.** Tendrá ambas mejoras disponibles.
- **Si es cualquier otra letra.** No tendrá mejoras disponibles.

Modificando para cada opción el diccionario de acciones del agente, por último, se creará la lista de prioridades en función de la introducida en el método `inputPriorities()`. Esta será la que se considera óptima (véase Ilustración 38) y para poder identificar este comportamiento se le asigna el nombre `Optimal` seguido de su Id.

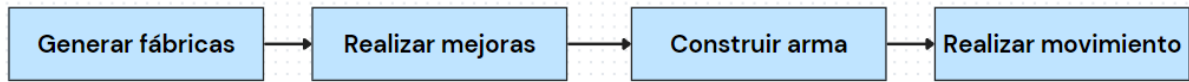


Ilustración 38. Lista de prioridad de los agentes óptimos

Una vez asignado el comportamiento personalizado, se debe optimizar cada acción para permitir que el agente obtenga el mayor número de puntos estelares posibles, para ello se modifica el método polimórfico `changeBehaviour()`. Con el objetivo de poder explicar el funcionamiento del método se comenta la forma de actuar para los distintos estados posibles que se encontrará el agente.

- **Si solo tiene planetas en casillas adyacentes.** Comprueba si tiene suficiente oro para mantener el planeta.
  - En el caso de poder mantenerlo, lo conquistará cambiando su dirección de movimiento hacia él y priorizando la acción de movimiento.
  - Si no puede mantenerlo, altera su lista de prioridad para poner como acción prioritaria la construcción de fábricas.
- **Si solo tiene agentes en casillas adyacentes.** Comprueba si tiene mejor arma que sus enemigos.
  - Si tiene mejor arma, luchará cambiando su dirección hacia ellos.
  - Si, por el contrario, tiene un arma peor, cambiará su lista de prioridades para poner como acción prioritaria el movimiento, permitiéndole escapar rápidamente de sus enemigos.
- **Si hay planetas y agentes en casillas adyacentes.** Si no tiene arma cambiara su lista de prioridad para priorizar la construcción de un arma.
- **Si no hay nada en casillas adyacentes.** Actuará según se haya especificado en la declaración de su lista de prioridades.

La Tabla 16 presenta los datos obtenidos tras ejecutar las 50 semillas del apartado 6.1.

Tabla 16. Resultados con comportamiento CustomBehaviour

Presencia Explorer	Presencia Chaser	Presencia Farmer	Presencia Custom
14 %	32 %	13 %	41 %
Turnos medios	Mediana de turnos	Tiempo medio	Mediana de tiempos
77.592	100.000	353 s	440 s

Cabe destacar que el número de agentes finales es el mayor en todas las simulaciones probadas contando con **1.277 agentes, un incremento del 78%** comparado con el sistema original. Con la incorporación de este comportamiento óptimo los porcentajes de presencia se han visto alterados, pasando este a ser el más abundante, demostrando que la funcionalidad implementada para obtener puntos estelares parece ser correcta y reduciendo los demás destacando la disminución de 13% en el comportamiento Explorer. A pesar de esta reducción, Explorer consigue la victoria en 16 ocasiones siendo el único comportamiento que lo consigue antes de los turnos límites con una media de 29.974 turnos. Por tanto, las victorias de Chaser (9) y CustomBehaviour (25) se dan siempre en los 100.000 turnos por ser los que más presencia tienen en el sistema, justificando el resultado de la mediana de turnos.

Adicionalmente, para las 16 simulaciones en las que el agente Explorer se corona como vencedor se decide realizar una pequeña modificación en el sistema. Cambiando el turno en el que se realiza la verificación para la eliminación y adición de un agente, pasando del turno 100 al turno 50. Con esta pequeña modificación, se observa que Explorer no consigue obtener ninguna victoria y el sistema se estabiliza llegando siempre a los 100.000 turnos sin un ganador claro. Lo que comprueba que, si se cambia el turno en el que se analiza el balance de los agentes, los resultados son considerablemente diferentes. Sin embargo, los porcentajes de mayor presencia son muy similares a los obtenidos en la Tabla 16, con la pequeña diferencia que Chaser se reduce al 29% y Farmer aumenta al 16%.

### 6.2.3. Agentes que comparten sus recursos (Friendly)

Este tipo de agentes presentan un comportamiento muy especial y es que evitan derrotar a sus enemigos en luchas. En su lugar persiguen al agente con menor balance en la simulación

(amigo) y le otorgan el 10% de su oro, el 5% de su tecnología y ambos reciben 1 punto estelar para evitar que sean eliminados, simulando así la aparición de una alianza entre estos agentes.

Para ello el amigo tendrá que estar en sus casillas adyacentes, lo que significa que, si tiene un arma, puede luchar con su donante “**Friendly**” y obtener nuevamente los recursos donados. De esta forma, se permite a los agentes que peor se están desarrollando en el sistema recuperar un cierto nivel económico y recuperar un balance de puntos positivo.

Para la implementación se genera la clase Friendly que hereda de Behaviour, añadiendo un atributo llamado “amigo”, que será el encargado de guardar la información del agente más necesitado. Además, se cambia dentro de ella su lista de prioridad (véase Ilustración 39), quitándole la opción de construir armas, la dirección de movimiento será hacia los planetas para permitirle obtener puntos estelares e idealmente no ser eliminado en los primeros 100 turnos. En cuanto a las mejoras, dispondrá de la capacidad de doblar los recursos generados por las fábricas. Este tipo de agente comenzará con 1.000 unidades de oro y tecnología y 5 fábricas para donar una mayor cantidad de oro desde el inicio y recuperar los recursos donados con las fábricas.

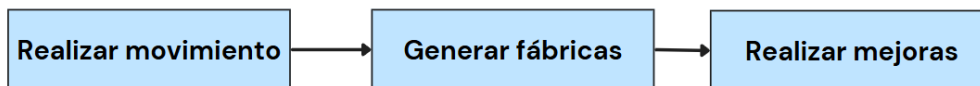


Ilustración 39. Lista de prioridades de Friendly

Para realizar la funcionalidad deseada, se modifica el método **changeBehaviour()**, cuya implementación queda representada en el diagrama de flujo de la Ilustración 40. En ella se reinicia el comportamiento para volver a identificar al “amigo” con menor balance, gracias al diccionario de enemigos (pasado como argumento a **changeBehaviour()**), convirtiéndolo en su objetivo de movimiento para acercarse con el método **addSpecialTarget()**. Si se encuentra en sus casillas adyacentes, le dona los recursos y le proporciona un punto estelar.



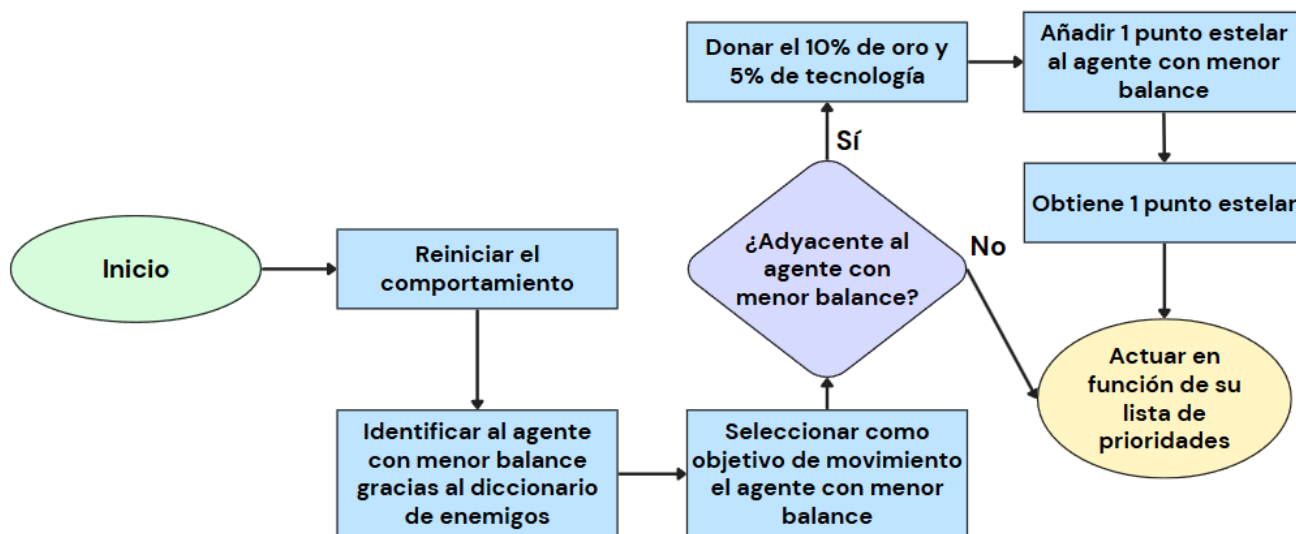


Ilustración 40. Diagrama de flujo de ChangeBehaviour() de la clase Friendly

Antes de comentar las pruebas realizadas, es necesario destacar un comportamiento emergente que surgió probando la funcionalidad de esta clase. En la Ilustración 41, se muestra la leyenda de un agente Friendly, donde se observa cómo el agente adquirió armas, permitiéndole luchar con los demás agentes. Por lo que, aunque les diera un porcentaje de sus beneficios, también los podía recuperar ganando su duelo. De esta forma pasó de un comportamiento de apoyo a una conducta que lo convertía en un “abusón” por combatir con los más necesitados.

Este cambio de conducta es posible debido a la forma de elegir las acciones, si los agentes obtienen un número menor a 0.02 realizarán una acción aleatoria entre todas las posibles. Por lo que, en este caso, el agente Friendly obtuvo un número más pequeño y seleccionó aleatoriamente la acción para la construcción de un arma.

■ **Agent:** 36 **Resources:** T: 2861 G: 2417 P: 0 F: 8 **Stellar Points:** -9 **Balance:** -9  
**Behaviour:** Friendly **Weapon:** Lasers **Battles Won:** 2 **Upgrades:** None

Ilustración 41. Leyenda de comportamiento emergente de Friendly

En cuanto a las pruebas, al tratarse de un comportamiento de apoyo no se espera que estos agentes consigan ninguna victoria, simplemente se estudiará cómo cambia el sistema al introducir un comportamiento que ayuda a los más necesitados. Para ello se modifica el método addAgent() de la clase Model, permitiendo la introducción de este tipo de agentes. Además, en



el turno 100 independientemente del agente introducido, se añadirá un agente Friendly, para que tras el primer ciclo de 100 turnos siempre haya al menos un agente de apoyo.

En la Tabla 17, se muestran los datos relevantes de las 50 ejecuciones de las semillas anteriores.

Tabla 17. Resultados con comportamiento Friendly

Presencia Explorer	Presencia Chaser	Presencia Farmer	Presencia Friendly
25 %	52 %	17 %	6 %
Turnos medios	Mediana de turnos	Tiempo medio	Mediana de tiempos
8.525	8.470	18 s	17 s

Los resultados de tiempo obtenidos son los más rápidos en todas las pruebas realizadas, esto se debe a que Friendly proporciona puntos estelares a su amigo, por lo que agiliza la obtención de puntos por parte de los jugadores, llegando antes al límite. Además, los agentes antiguos se aprovechan de este nuevo comportamiento para permanecer en el sistema, aumentando la diferencia de puntos con los nuevos agentes. Esto hace que el resultado de agentes finales sea de **670**, lo que supone **una reducción del 6% respecto al sistema original**.

Por otro lado, se decide probar cómo funcionaría esta funcionalidad si el sistema modificara el turno en el que se comprueba el balance de los agentes para ser eliminados y añadidos. La Tabla 18 recoge los resultados obtenidos para la revisión cada 50 turnos y la Tabla 19 para cada 200 turnos.

Tabla 18. Resultados con comportamiento Friendly cada 50 turnos

Presencia Explorer	Presencia Chaser	Presencia Farmer	Presencia Friendly
22 %	47 %	24 %	7 %
Turnos medios	Mediana de turnos	Tiempo medio	Mediana de tiempos
11.901	9.167	27 s	19 s

Se observa que al revisar el balance de los agentes cada 50 turnos, estos aumentan al compararlos con la Tabla 17 en un 39%, mientras que la mediana de los turnos aumenta solo un 8%. Esta diferencias es debida a la existencia de ejecuciones anómalas, donde algunos

jugadores que normalmente no serían eliminados se quedan fuera del sistema, aumentando los turnos y en su defecto los tiempos. Lo que supone una reducción de agentes finales (643) del 4% en comparación con el sistema con Friendly. Además, en esta ocasión los agentes Farmer han conseguido un 7% más de presencia en el sistema comparado con los datos de la Tabla 17.

Tabla 19. Resultados con comportamiento Friendly cada 200 turnos

Presencia Explorer	Presencia Chaser	Presencia Farmer	Presencia Friendly
22 %	54 %	20 %	4 %
Turnos medios	Mediana de turnos	Tiempo medio	Mediana de tiempos
9.394	8.547	20 s	17 s

Al realizar la comprobación cada 200 turnos el aumento de turnos es mínimo y en la mayoría de los ganadores se corresponden con agentes con id 1, ya que logran sobrevivir más tiempo en el sistema, al disponer de más turnos para revertir el balance negativo. Sin embargo, el número de agentes finales aumenta en esta ocasión pasando de los 670 del sistema con Friendly a 743 agentes, **un aumento del 11%**. Además, la presencia de Friendly es la menor en las tres pruebas realizadas llegando solo al 4% de presencia final.

Para concluir este capítulo, se ejecutan nuevamente las 50 semillas utilizadas, pero con la diferencia que esta vez se ejecutarán de manera simultánea las 3 mutaciones comentadas anteriormente. Con el objetivo de determinar si se obtiene algún resultado diferente a los obtenidos de manera individual o si siguen el mismo patrón. Las pruebas finalizan con un total de **1052 agentes finales** y los porcentajes de aparición muestran al **comportamiento óptimo** como el más abundante con un **42%**, demostrando que es capaz de sobrevivir independientemente de los agentes introducidos en el sistema. Sin embargo, la distribución de ganadores representa una muestra de los datos obtenidos en cada modificación individual, ya que **Explorer (12% presencia)** sigue ganado la gran mayoría. Por otro lado, **RandomBehaviour (8% presencia)** consigue obtener la victoria en 2 simulaciones con la lista de prioridades considerada óptima y **Friendly (3% presencia)**, aunque no gane, consigue ayudar al máximo número de agentes para llegar antes a los puntos límites, acortando los turnos y tiempos de ejecución. Por su parte, **Chaser** es el segundo comportamiento más presente con **26%** y **Farmer** finaliza con un **9% de aparición**.













## Capítulo 7. Conclusión

Al finalizar la iteración 7 se considera como terminado todo el proyecto, contando con un sistema que cumple con todos los objetivos propuestos al principio del TFG.

- Se ha conseguido entender todas las posibilidades para la generación de un ABM en el lenguaje Python.
- Se ha creado un sistema de reglas bastante más complejo de lo deseado, pero que permite tener varios comportamientos diversos.
- Se ha logrado implementar todo el sistema en Python gracias al framework Mesa.
- Se han estudiado y analizado distintos comportamientos en el capítulo 6.

A través de este estudio, se demuestra cómo el sistema es capaz de soportar la introducción dinámica de nuevas conductas que no habían sido programados explícitamente en el agente, balanceando el sistema para obtener nuevos resultados que difieren de los esperados, independientemente de si estos cambios resultan ventajosos o perjudiciales para los agentes. Permitiendo que el sistema se ajuste lo máximo posible al comportamiento que tendría un modelo del mundo real. A través de la Tabla 20, se estudian los porcentajes de variación en el sistema en función del cambio dinámico introducido, comparados con el sistema original.

Tabla 20. Comparación de ejecuciones con el sistema original

Tipo de ejecución	Presencia Explorer	Presencia Chaser	Presencia Farmer	Agentes Finales
Por defecto	27%	54%	19%	716
Cambio en todos los Explorer	 -4 %	 +3 %	 +1 %	 +20 %
Cambio en todos los Chaser	 +5 %	 -25 %	 +20 %	 +64 %
Cambio en todos los Farmer	 -9 %	 -23 %	 +32 %	 +54 %

Estos cambios dinámicos se llevan a cabo de una manera sencilla gracias al lenguaje interpretado utilizado (Python) y a los métodos polimórficos desarrollados para que el sistema funcione independientemente de los comportamientos de los agentes. Facilitando, de esta forma, la introducción en un futuro de un algoritmo genético que espontáneamente genere nuevas mutaciones. Además, se ha podido contemplar un comportamiento emergente propio del sistema, sin necesidad de incorporarlo dinámicamente. Como es el caso de la conducta Friendly, que consigue desarrollar un arma y cambiar drásticamente su funcionalidad dentro del sistema, atacando a los agentes a los que donaba sus recursos.

Por otra parte, las modificaciones introducidas en el apartado 6.2 demuestran cómo el sistema es robusto permitiendo la ejecución de numerosas simulaciones, destacando los 353 segundos medios para el comportamiento más exigente CustomBehaviour. Además, el buen desempeño de RandomBehaviour, que casi iguala los resultados ganadores de Explorer, manifiesta que aún existen numerosas mutaciones óptimas que podrían superar a las ya probadas.

## 7.1. Pasos futuros

El código generado funciona de manera correcta, sin embargo, el sistema presentado no es perfecto y se puede seguir trabajando en él para obtener un proyecto más completo. Las principales líneas de trabajo que se pueden seguir, tras este TFG son las siguientes:

- Añadir nuevas formas de obtención de puntos estelares, para favorecer otras estrategias y así balancear el sistema de manera diferente.
- Implementar un algoritmo genético, que permita generar de una manera más orgánica las mutaciones de los agentes.
- Recuperar la idea de implementar un algoritmo de aprendizaje por refuerzo, para que los agentes puedan obtener las acciones realmente óptimas para cada situación.
- Observar nuevos comportamientos emergentes en los agentes.
- Crear una herramienta más sofisticada para el guardado y cargado de ejecuciones.
- Entender el funcionamiento completo de la interfaz gráfica de Mesa, para poder modificarla y añadir funcionalidades extra.

## Bibliografía

- [1]. Macal, C. & North, Michael. (2008). Agent-based modeling and simulation: ABMS examples. *Proceedings - Winter Simulation Conference*. 101-112. <https://doi.org/10.1109/WSC.2008.4736060>
- [2]. De Marchi, S., & Page, S. E. (2014). Agent-based models. *Annual Review of political science*, 17(1), 1-20. <https://doi.org/10.1146/annurev-polisci-080812-191558>
- [3]. Eugene M. Izhikevich et al. (2015) Game of Life. *Scholarpedia*, 10(6):1816. <https://doi.org/10.4249/scholarpedia.1816>
- [4]. Clarke, Keith. (2014). Cellular Automata and Agent-Based Models. [https://doi.org/10.1007/978-3-642-23430-9\\_63](https://doi.org/10.1007/978-3-642-23430-9_63)
- [5]. Kasereka, S. K., Zohinga, G. N., Kiketa, V. M., Ngoie, R. B. M., Mputu, E. K., Kasoro, N. M., & Kyandoghere, K. (2023). Equation-based modeling vs. agent-based modeling with applications to the spread of COVID-19 outbreak. *Mathematics*, 11(1), 253. <https://doi.org/10.3390/math11010253>
- [6]. Mata, A.S., Dourado, S.M.P. Mathematical modeling applied to epidemics: an overview. *São Paulo J. Math. Sci.* **15**, 1025–1044 (2021). <https://doi.org/10.1007/s40863-021-00268-7>
- [7]. Aguiar, M., Anam, V., Blyuss, K. B., Estadilla, C. D. S., Guerrero, B. V., Knopoff, D., ... & Stollenwerk, N. (2022). Mathematical models for dengue fever epidemiology: A 10-year systematic review. *Physics of Life Reviews*, 40, 65-92. <https://doi.org/10.1016/j.plrev.2022.02.001>
- [8]. Tang, B., Wang, X., Li, Q., Bragazzi, N. L., Tang, S., Xiao, Y., & Wu, J. (2020). Estimation of the Transmission Risk of the 2019-nCoV and Its Implication for Public Health Interventions. *Journal of clinical medicine*, 9(2), 462. <https://doi.org/10.3390/jcm9020462>
- [9]. Hunter, E., Mac Namee, B. & Kelleher, J. D. (2020). A hybrid agent-based and equation based model for the spread of infectious diseases. *The Journal of Artificial Societies and Social Simulation* vol. 23(4), pages 1-14. <https://doi.org/10.18564/jasss.4421>
- [10]. Bostanci, I., & Conrad, T. (2024). Integrating Agent-Based and Compartmental Models for Infectious Disease Modeling: A Novel Hybrid Approach. *arXiv preprint*. <https://doi.org/10.48550/arXiv.2407.20993>
- [11]. Qiao, Q., Cheung, C., Yunusa-Kaltungo, A., Manu, P., Cao, R., & Yuan, Z. (2023). An interactive agent-based modelling framework for assessing COVID-19 transmission risk on construction site. *Safety Science*, 168 (106312). <https://doi.org/10.1016/j.ssci.2023.106312>
- [12]. Brusatin, S., Padoan, T., Coletta, A., Gatti, D. D., & Glielmo, A. (2024). Simulating the economic impact of rationality through reinforcement learning and agent-based modelling. *arXiv (Cornell University)*. <https://doi.org/10.48550/arxiv.2405.02161>

- [13]. Damiani, G. M. (2024). An essay on the history of DSGE models. *arXiv preprint arXiv*. <https://doi.org/10.48550/arXiv.2409.00812>.
- [14]. Szczepanska, T., Antosz, P., Berndt, J. O., Borit, M., Chattoe-Brown, E., Mehryar, S., ... Verhagen, H. (2022). GAM on! Six ways to explore social complexity by combining games and agent-based models. *International Journal of Social Research Methodology*, 25(4), 541–555. <https://doi.org/10.1080/13645579.2022.2050119>
- [15]. Salvini, G., Ligtenberg, A., van Paassen, A., Bregt, A. K., Avitabile, V., & Herold, M. (2016). REDD+ and climate smart agriculture in landscapes: A case study in Vietnam using companion modelling. *Journal of Environmental Management*, 172, 58–70. <https://doi.org/10.1016/j.jenvman.2015.11.060>
- [16]. Gomes, S., Dias, J., & Martinho, C. (2019). GIMME: Group Interactions Manager for Multiplayer sErious games. *2019 IEEE Conference on Games (CoG)*, 2019, 1–8. <https://doi.org/10.1109/CIG.2019.8847962>
- [17]. Belem, M., Bazile, D., & Coulibaly, H. (2018). Simulating the Impacts of Climate Variability and Change on Crop Varietal Diversity in Mali (West-Africa) Using Agent-Based Modeling Approach. *Journal of Artificial Societies and Social Simulation*, 21(2). <https://doi.org/10.18564/jasss.3690>
- [18]. Cedeno-Mieles, V., Hu, Z., Ren, Y., Deng, X., Adiga, A., Barrett, C., ... Self, N. (2020). Networked experiments and modeling for producing collective identity in a group of human subjects using an iterative abduction framework. *Social Network Analysis and Mining*, 10(1), 11. <https://doi.org/10.1007/s13278-019-0620-8>
- [19]. Yang, L., Zhang, L., Philippopoulos-Mihalopoulos, A., Chappin, E. J. L., & van Dam, K. H. (2020). Integrating agent-based modeling, serious gaming, and co-design for planning transport infrastructure and public spaces. *URBAN DESIGN International*. <https://doi.org/10.1057/s41289-020-00117-7>
- [20]. Kikuchi, T., Tanaka, Y., Kunigami, M., Yamada, T., Takahashi, H., & Terano, T. (2019). Debriefing Framework for Business Games Using Simulation Analysis. *Communications in Computer and Information Science*, 999(1), 64–76. [https://doi.org/10.1007/978-981-13-6936-0\\_8](https://doi.org/10.1007/978-981-13-6936-0_8)
- [21]. Antelmi, A., Cordasco, G., D'Ambrosio, G., De Vinco, D., & Spagnuolo, C. (2022). Experimenting with Agent-Based Model Simulation Tools. *Applied Sciences*, 13(1), 13. <https://doi.org/10.3390/app13010013>
- [22]. Foramitti, J., (2021). AgentPy: A package for agent-based modeling in Python. *Journal of Open Source Software*, 6(62), 3065, <https://doi.org/10.21105/joss.03065>
- [23]. Collier, Nicholson, and Jonathan Ozik. (2022). “Distributed Agent-Based Simulation with Repast4Py.” In 2022 Winter Simulation Conference (WSC), 192–206. Singapore: IEEE. <https://doi.org/10.1109/WSC57314.2022.10015389>.

- [24]. Jaxa-Rozen, Marc & Kwakkel, Jan. (2018). PyNetLogo: Linking NetLogo with Python. *Journal of Artificial Societies and Social Simulation*. 21. <https://doi.org/10.18564/jasss.3668>
- [25]. Rubio-Campillo, Xavier. (2014). Pandora: A Versatile Agent-Based Modelling Platform for Social Simulation. <https://doi.org/10.13140/2.1.5149.4086>.
- [26]. Masad, D., & Kazil, J. L. (2015, July). Mesa: An Agent-Based Modeling Framework. In *SciPy* (pp. 51-58).
- [27]. *Repositorio de Github* [en línea]. (20 de agosto de 2024). Recuperado de <https://github.com/projectmesa/mesa-examples/tree/main/examples>
- [28]. *Definición de 4X* [en línea]. (25 de marzo de 2024). Recuperado de <https://www.devuego.es/gamerdic/termino/4x>
- [29]. *Código de la clase RandomActivationByTypeFiltered* [en línea]. (29 de agosto de 2024). Código recuperado de [https://github.com/projectmesa/mesa-examples/blob/fbf80af30d4ed6dce01bee54020b8c15cfb3ae8f/examples/wolf\\_sheep/wolf\\_sheep/scheduler.py](https://github.com/projectmesa/mesa-examples/blob/fbf80af30d4ed6dce01bee54020b8c15cfb3ae8f/examples/wolf_sheep/wolf_sheep/scheduler.py)
- [30]. Definición lenguajes interpretados y ejemplos [en línea]. (2024) Recuperado de <https://www.unir.net/revista/ingenieria/lenguajes-programacion-interpretados/>
- [31]. Abid All Awan (2022). *An Introduction to Q-Learning: A Tutorial For Beginners*. Disponible en: <https://www.datacamp.com/tutorial/introduction-q-learning-beginner-tutorial>





## Anexo

### Anexo A. Código del proyecto

Enlace a repositorio de Github con todo el código comentado en este trabajo.

- <https://github.com/Dabyz24/AgentBasedModeling4xGame>

### Anexo B. Análisis del proyecto

Enlace a archivo Excel, donde se presentan los resultados estudiados para todos los comportamientos, divididos en distintas hojas.

- [https://github.com/Dabyz24/AgentBasedModeling4xGame/blob/main/Analisis\\_resultados.xlsx](https://github.com/Dabyz24/AgentBasedModeling4xGame/blob/main/Analisis_resultados.xlsx)

Carpeta “saves” con todas las simulaciones guardadas.

- <https://github.com/Dabyz24/AgentBasedModeling4xGame/tree/main/saves>

### Anexo C. Ilustraciones de ejecuciones del punto 6.1.

#### Ejecución de cambio dinámico en Explorer

- Leyenda turno 20.

■ **Agent: 1** Resources: T: 485 G: 91 P: 2 F: 10 **Stellar Points: 26** Balance: 26  
Behaviour: **Explorer** Weapon: **N** Battles Won: 0 Upgrades: **None**

---

■ **Agent: 2** Resources: T: 12 G: 32 P: 1 F: 0 **Stellar Points: 3** Balance: 3  
Behaviour: **Chaser** Weapon: **Plasma Cannon** Battles Won: 2 Upgrades: **None**

---

■ **Agent: 3** Resources: T: 53 G: 132 P: 1 F: 7 **Stellar Points: 2** Balance: 2  
Behaviour: **Farmer** Weapon: **Plasma Cannon** Battles Won: 2 Upgrades: **None**

- Leyenda turno 48.

■ **Agent: 1** Resources: T: 2469 G: 1820 P: 6 F: 10 **Stellar Points: 174** Balance: 174  
Behaviour: **DummyExplorer** Weapon: **N** Battles Won: 0 Upgrades: **None**

---

■ **Agent: 2** Resources: T: 147 G: 163 P: 1 F: 0 **Stellar Points: 32** Balance: 32  
Behaviour: **Chaser** Weapon: **Guided missil** Battles Won: 4 Upgrades: **D ↑**

---

■ **Agent: 3** Resources: T: 378 G: 439 P: 0 F: 14 **Stellar Points: 17** Balance: 17  
Behaviour: **Farmer** Weapon: **Guided missil** Battles Won: 2 Upgrades: **None**

---

- Leyenda turno 115.

■ **Agent: 1** Resources: T: 898 G: 734 P: 0 F: 11 **Stellar Points: 239** Balance: -5  
Behaviour: **DummyExplorer** Weapon: **N** Battles Won: 0 Upgrades: **None**

---

■ **Agent: 2** Resources: T: 5756 G: 4840 P: 5 F: 2 **Stellar Points: 222** Balance: 73  
Behaviour: **Chaser** Weapon: **Guided missil** Battles Won: 42 Upgrades: **D ↑**

---

■ **Agent: 3** Resources: T: 903 G: 873 P: 0 F: 22 **Stellar Points: 81** Balance: 5  
Behaviour: **Farmer** Weapon: **Guided missil** Battles Won: 6 Upgrades: **None**

---

■ **Agent: 4** Resources: T: 166 G: 115 P: 1 F: 0 **Stellar Points: 9** Balance: 9  
Behaviour: **Chaser** Weapon: **Guided missil** Battles Won: 3 Upgrades: **None**

---

### Ejecución cambio dinámico en Chaser

- Leyenda simplificada turno 4299. Último turno de agente Agresive en la semilla 1390 con el cambio dinámico en el turno 50.

■ **Agent: 2** Resources: T: 24840 G: 18057 P: 0 F: 32 **Stellar Points: 3463** Balance: -5  
Behaviour: **Agressive** Weapon: **Guided missil** Battles Won: 1759 Upgrades: D ↑

---

■ **Agent: 16** Resources: T: 28783 G: 28461 P: 1 F: 46 **Stellar Points: 781** Balance: 3  
Behaviour: **Chaser** Weapon: **Guided missil** Battles Won: 2427 Upgrades: D ↑

---

■ **Agent: 22** Resources: T: 26230 G: 19121 P: 0 F: 45 **Stellar Points: 728** Balance: -13  
Behaviour: **Chaser** Weapon: **Guided missil** Battles Won: 2025 Upgrades: D ↑

---

■ **Agent: 28** Resources: T: 25784 G: 19381 P: 0 F: 45 **Stellar Points: 782** Balance: 11  
Behaviour: **Chaser** Weapon: **Guided missil** Battles Won: 1388 Upgrades: D ↑

---

■ **Agent: 30** Resources: T: 25619 G: 19700 P: 0 F: 45 **Stellar Points: 684** Balance: 0  
Behaviour: **Chaser** Weapon: **Guided missil** Battles Won: 1376 Upgrades: D ↑

---

■ **Agent: 31** Resources: T: 33110 G: 36738 P: 0 F: 40 **Stellar Points: 2798** Balance: 39  
Behaviour: **Explorer** Weapon: **Guided missil** Battles Won: 323 Upgrades: None

---

■ **Agent: 36** Resources: T: 44290 G: 52843 P: 0 F: 44 **Stellar Points: 451** Balance: 23  
Behaviour: **Chaser** Weapon: **Guided missil** Battles Won: 820 Upgrades: D ↑

---

■ **Agent: 38** Resources: T: 50803 G: 54302 P: 0 F: 42 **Stellar Points: 295** Balance: 23  
Behaviour: **Chaser** Weapon: **Guided missil** Battles Won: 771 Upgrades: D ↑

---

■ **Agent: 39** Resources: T: 16129 G: 17084 P: 0 F: 35 **Stellar Points: 442** Balance: 39  
Behaviour: **Explorer** Weapon: **Guided missil** Battles Won: 175 Upgrades: None

---

■ **Agent: 40** Resources: T: 18285 G: 18213 P: 0 F: 41 **Stellar Points: 27** Balance: 18  
Behaviour: **Farmer** Weapon: **Guided missil** Battles Won: 212 Upgrades: F ↑

---

■ **Agent: 41** Resources: T: 38135 G: 38811 P: 0 F: 42 **Stellar Points: 99** Balance: 43  
Behaviour: **Chaser** Weapon: **Guided missil** Battles Won: 370 Upgrades: D ↑

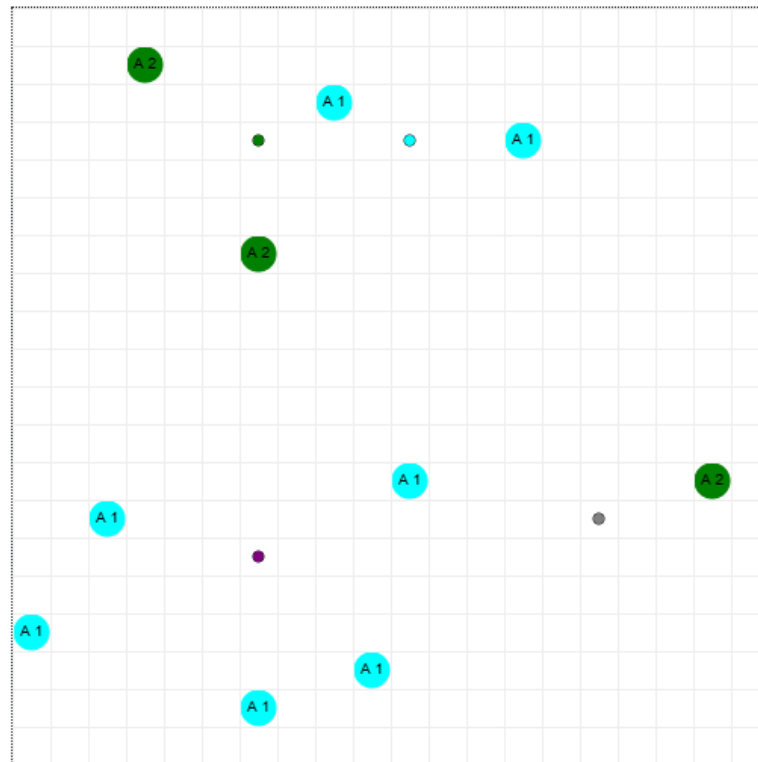
---

■ **Agent: 42** Resources: T: 50476 G: 56765 P: 0 F: 42 **Stellar Points: 69** Balance: 37  
Behaviour: **Chaser** Weapon: **Guided missil** Battles Won: 345 Upgrades: D ↑

### Ejecución cambio dinámico en Farmer

- Estado del sistema sin ninguna modificación en el turno 199. Último turno de agente Farmer en la semilla 4472.

Current Step: 199



Total players: 4

■ **Agent: 1** Resources: T: 7368 G: 2784 P: 7 F: 26 **Stellar Points: 645** Balance: 421  
 Behaviour: **Explorer** Weapon: **N** Battles Won: 0 Upgrades: **None**

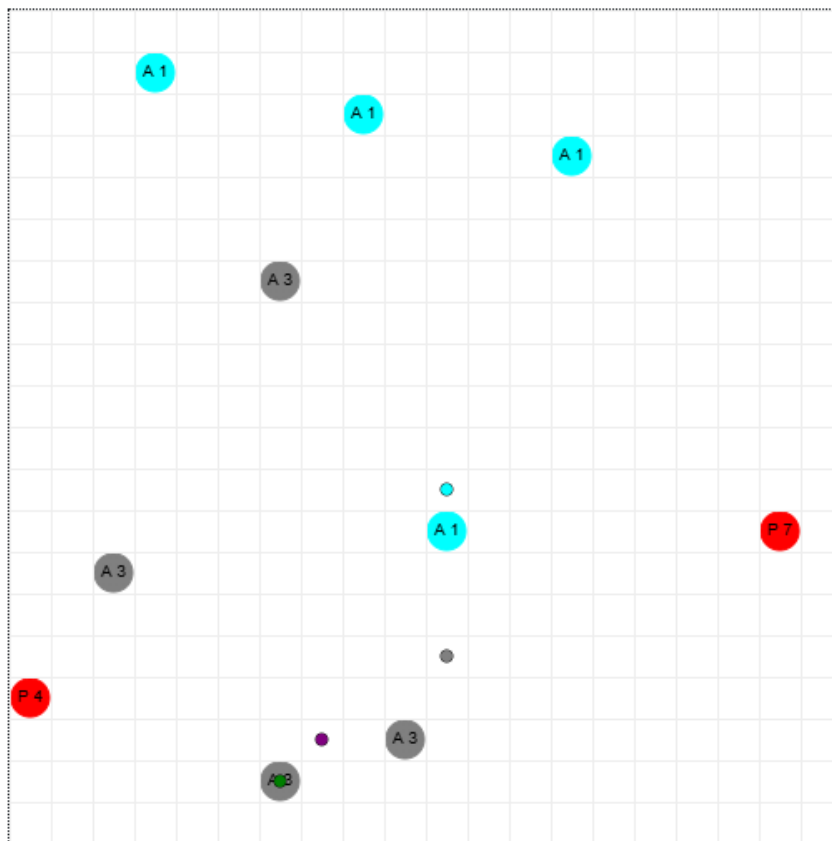
■ **Agent: 2** Resources: T: 3350 G: 5484 P: 3 F: 21 **Stellar Points: 200** Balance: 102  
 Behaviour: **Chaser** Weapon: **Guided missile** Battles Won: 58 Upgrades: **D 1**

■ **Agent: 3** Resources: T: 1672 G: 3546 P: 0 F: 26 **Stellar Points: 77** Balance: -9  
 Behaviour: **Farmer** Weapon: **Guided missile** Battles Won: 25 Upgrades: **F 1**

■ **Agent: 4** Resources: T: 469 G: 378 P: 0 F: 17 **Stellar Points: 138** Balance: 138  
 Behaviour: **Explorer** Weapon: **N** Battles Won: 0 Upgrades: **None**

- Estado del sistema con modificación dinámica en el turno 199 en la semilla 4472.

Current Step: 199



Total players: 4

■ **Agent: 1** Resources: T: 2497 G: 1485 P: 4 F: 25 **Stellar Points: 341** Balance: 235  
 Behaviour: **Explorer** Weapon: **N** Battles Won: 0 Upgrades: **None**

■ **Agent: 2** Resources: T: 1391 G: 879 P: 0 F: 23 **Stellar Points: 89** Balance: 31  
 Behaviour: **Chaser** Weapon: **Guided missile** Battles Won: 78 Upgrades: **D ↑**

■ **Agent: 3** Resources: T: 4338 G: 3306 P: 4 F: 27 **Stellar Points: 472** Balance: 318  
 Behaviour: **OptimalFarmer** Weapon: **Guided missile** Battles Won: 47 Upgrades: **D ↑ F ↑**

■ **Agent: 4** Resources: T: 1614 G: 521 P: 0 F: 17 **Stellar Points: 1** Balance: 1  
 Behaviour: **Chaser** Weapon: **Guided missile** Battles Won: 43 Upgrades: **D ↑**

- Leyenda del sistema sin modificar en el turno 575 en la semilla 4472.

■ **Agent: 1** Resources: T: 3176 G: 4082 P: 0 F: 32 **Stellar Points: 1478** Balance: 107  
Behaviour: **Explorer** Weapon: **Lasers** Battles Won: 21 Upgrades: **None**

---

■ **Agent: 2** Resources: T: 17450 G: 22243 P: 0 F: 32 **Stellar Points: 1013** Balance: 64  
Behaviour: **Chaser** Weapon: **Guided missil** Battles Won: 306 Upgrades: **D ↑**

---

■ **Agent: 4** Resources: T: 7715 G: 3735 P: 8 F: 31 **Stellar Points: 1011** Balance: 424  
Behaviour: **Explorer** Weapon: **Plasma Cannon** Battles Won: 2 Upgrades: **None**

---

■ **Agent: 5** Resources: T: 1085 G: 2324 P: 0 F: 28 **Stellar Points: 375** Balance: -20  
Behaviour: **Explorer** Weapon: **N** Battles Won: 0 Upgrades: **None**

---

■ **Agent: 6** Resources: T: 13100 G: 14603 P: 1 F: 31 **Stellar Points: 133** Balance: 43  
Behaviour: **Chaser** Weapon: **Guided missil** Battles Won: 175 Upgrades: **D ↑**

---

■ **Agent: 8** Resources: T: 4802 G: 9523 P: 0 F: 9 **Stellar Points: -9** Balance: -9  
Behaviour: **Farmer** Weapon: **Plasma Cannon** Battles Won: 28 Upgrades: **None**

- Leyenda del sistema modificando la clase Farmer a nivel global en la semilla 4472.

■ **Agent: 2** Resources: T: 11368 G: 9040 P: 1 F: 34 **Stellar Points: 340** Balance: 83  
Behaviour: **Chaser** Weapon: **Guided missil** Battles Won: 302 Upgrades: **D ↑**

---

■ **Agent: 3** Resources: T: 13206 G: 18887 P: 1 F: 36 **Stellar Points: 1365** Balance: 38  
Behaviour: **Farmer** Weapon: **Guided missil** Battles Won: 199 Upgrades: **D ↑ F ↑**

---

■ **Agent: 6** Resources: T: 9460 G: 8293 P: 0 F: 34 **Stellar Points: 783** Balance: 45  
Behaviour: **Farmer** Weapon: **Guided missil** Battles Won: 110 Upgrades: **D ↑ F ↑**

---

■ **Agent: 7** Resources: T: 7909 G: 3307 P: 3 F: 30 **Stellar Points: 352** Balance: 221  
Behaviour: **Farmer** Weapon: **Guided missil** Battles Won: 20 Upgrades: **D ↑ F ↑**

---

■ **Agent: 8** Resources: T: 700 G: 732 P: 5 F: 17 **Stellar Points: 121** Balance: 121  
Behaviour: **Farmer** Weapon: **Guided missil** Battles Won: 0 Upgrades: **D ↑**

---