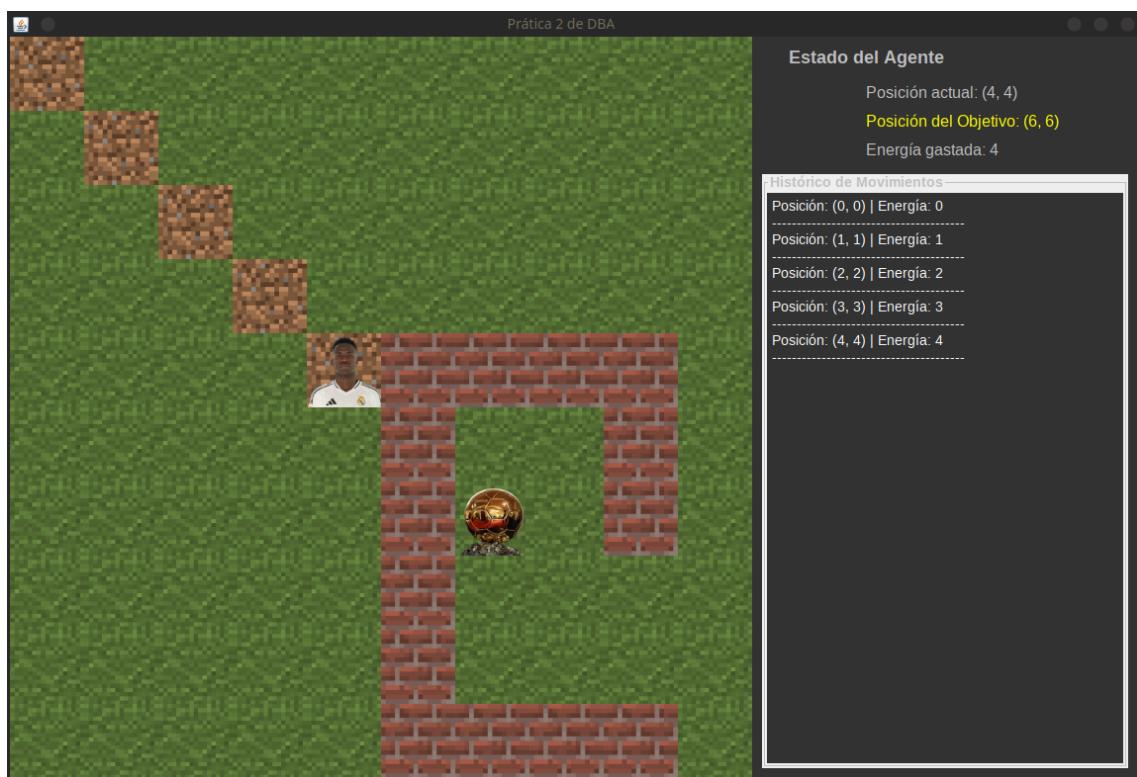


## Práctica 2: Movimiento de un agente en un mundo bidimensional

Equipo de prácticas 209

Fecha de entrega: 21 de noviembre de 2024



# 1. Introducción

Este documento es una memoria del desarrollo de la Práctica, en el cual se detallan decisiones de diseño e implementación que han dado como resultado el estado actual de esta práctica en el momento de la entrega.

En la práctica, se aplica el patrón de funcionamiento del framework de Java llamado JADE, enfocado en crear sistemas multiagentes. Hemos desarrollado un agente reactivo “inteligente” que es capaz de desplazarse por un mundo de dos dimensiones evitando obstáculos y dirigiéndose hacia una posición objetivo. Este agente contará con un estado interno conformado por una “memoria” que le permitirá alcanzar su objetivo en los diferentes mapas del simulador.

## 2. Objetivos

- Aplicar el framework JADE para la implementación de un agente reactivo “inteligente”.
- Aplicar buenas prácticas de programación.
- Aplicar patrones de diseño pertinentes de una forma adecuada.
- Crear un agente con estados internos capaz de alcanzar una casilla objetivo.
- Crear un agente que sea capaz de evitar obstáculos horizontales, verticales, diagonales, cónicos y convexos.
- Optimizar el comportamiento del agente para que consuma el menor porcentaje de energía posible, es decir, que alcance la casilla objetivo en el menor número de pasos (casillas transitadas) posible.

## 3. Decisiones de diseño

A continuación, se presentan los diferentes patrones de diseño aplicados a la arquitectura de nuestro software.

### 3.1. Patrón *Model-view-controller (MVC)* para la arquitectura general del software

Es un modelo de diseño arquitectónico utilizado para organizar y estructurar aplicaciones. Se divide en tres componentes principales, que separan la lógica de negocio, la presentación y la interacción del usuario.

#### Modelo:

Es la capa que gestiona los datos, la lógica de negocio y las reglas de la aplicación. En este caso particular, el modelo está compuesto por las clases Entorno, Mapa, Posicion, el enumerado Accion, así como las clases del paquete “sensores”: Sensor, Energia, Vision; y, por supuesto, la clase Agente, ubicada en el paquete .“agentes”.

Este notifica tanto a las vistas como a los sensores cuando hay cambios en el Entorno, de manera que se actualicen en consecuencia. En el caso de las vistas, estas se actualizarán de manera que reflejen los nuevos movimientos del agente en el mapa y, en el caso de los sensores, estos se actualizarán para contener los datos actualizados acerca del entorno para que el agente los pueda utilizar para decidir el próximo movimiento.

#### Vista:

Es la capa encargada de la presentación. Muestra los datos al usuario y gestiona la interfaz gráfica. No contiene lógica de negocio, solo se centra en cómo se presentan los datos. Escucha al modelo para actualizar la interfaz cuando los datos cambian. Nosotros contamos con dos vistas diferenciadas (dentro del paquete “vista”): VistaTexto, la cual refleja el simulador de manera textual, es decir, mediante la consola (terminal); y VistaGrafica, la cual refleja de manera gráfica el simulador, haciendo uso de otras clases creadas a tal efecto, junto con el uso de la biblioteca gráfica

de Java llamada "Swing", basada en componentes.

Las otras clases, ubicadas en el mismo paquete "vista", son PanelMapa, la cual representa el panel del mapa y muestra cada celda de una manera gráfica distinta. Esta clase es usada por VistaGrafica, conteniendo una instancia de esta, a modo de panel con el mapa del simulador (VistaGrafica sería la ventana principal del simulador, que contiene el panel anterior con el mapa). La otra clase que falta por comentar es PanelConfiguracionControlador, la cual representa un panel para la configuración del controlador. Permite seleccionar el mapa para la simulación, así como la posición inicial del agente y la posición de la casilla objetivo. Básicamente, permite configurar los datos básicos para la simulación de forma gráfica, encargándose en última instancia de crear el controlador que lanzará al agente en base a los datos introducidos en este panel por parte del usuario.

#### Controlador:

Es la capa que actúa como intermediaria con el modelo. Se encarga de usar el modelo solo a través de esta capa, evitando que se pueda manipular de forma errónea la funcionalidad de la lógica de negocio. En nuestro caso, tenemos un paquete "controlador", el cual contiene una única clase Controlador, que se encarga de lanzar al agente de la simulación.

La lógica de nuestro Controlador es la que sigue:

- Obtener la instancia del entorno de ejecución.
- Configurar el perfil para el contenedor principal.
- Crear el contenedor de agentes.
- Crear el agente estableciendo su nombre, la clase que lo representa y pasando el Entorno al agente para que este pueda interactuar con el mapa, entre otros aspectos.

### **3.2. Patrón *Singleton* para el entorno del agente**

Es un patrón de diseño creacional que garantiza que una clase tenga una única instancia en toda la aplicación y proporciona un punto global de acceso a ella.

Hemos aplicado este patrón a la clase Entorno, por las siguientes razones principales:

- Solo puede haber un entorno de simulación.
- Se quiere un estado del entorno para poder ser usado de manera consistente y son problemas colaterales en diferentes partes de la arquitectura del software del simulador.

Para conseguir aplicar este patrón, hemos realizado lo siguiente a nivel de código:

- Crear un atributo de clase con la única instancia que se va a gestionar de la propia clase.
- Ofrecer un método estático para obtener dicha instancia, asegurando que todas las partes del código accedan a la misma. Este método comprueba cuando es llamado si la instancia aún no existe, en cuyo caso se crea la primera vez que se solicita. Para el resto de llamadas, devuelve siempre la misma instancia.
- El constructor es privado, para evitar instanciación directa.

### **3.3. Patrón *Observer* para el entorno del agente, los sensores y las vistas**

Es un patrón de diseño comportamental que define una relación de dependencia uno a muchos entre objetos, de forma que cuando un objeto cambia su estado, notifica automáticamente a todos los objetos dependientes (observadores).

Aquí se identifican dos roles: el observador y el observado. La idea es que una serie de objetos hacen el papel de observadores y un objeto en particular hace el papel de observado. En nuestro caso, hemos creado un *Observer* un poco más complejo de lo normal, pues en el entramado del simulador, tenemos dos tipos de objetos observadores, los cuales son los sensores y las vistas. Todos

ellos observan al Entorno, que es la clase observada. Todos los observadores se suscriben automáticamente cuando se crean, pues reciben una instancia del entorno, la cual utilizan para añadirse al vector de ellos que contiene el entorno para que este, cuando realice cambios importantes, actualice en consecuencia todos estos objetos recorriéndolos uno a uno. El Entorno también dispone de un método para permitir que los objetos puedan darse de baja de las notificaciones.

Los observadores tienen una clase abstracta como base, la cual proporciona la cabecera de un método para actualizar, el cual tendrá que ser implementado por cada sensor o vista en específico. De esta manera, se podrá hacer uso del polimorfismo cuando el Entorno notifique las actualizaciones a los observadores.

Para arrojar una mayor claridad, especificamos a continuación el flujo básico para el funcionamiento de este patrón en nuestro simulador:

- Crear los sensores y las vistas, los cuales se suscriben automáticamente con la instancia del Entorno que reciben en el constructor, haciendo uso de los métodos del Entorno `registrarSensorz` `registrarVista`".
- El Entorno, cuando es ejecutado su método `.^actualizarPercepciones`", el cual recibe la nueva posición del agente y actualiza su estado en el mapa, notifica tanto a los sensores como a las vistas, haciendo uso de los métodos de este `"notificarSensoresz"` `"notificarVistas"`. De esta manera, al final de la lógica del método `.^actualizarPercepciones`", tras haber llevado a cabo los cambios importantes, actualiza cada objeto observador suscrito previamente, pues los tiene almacenados en sendos arrays de esos objetos.

Hemos aplicado este patrón a la clase Entorno, junto con los sensores y las vistas, por las siguientes razones principales:

- Automatizar la actualización de las vistas y los sensores.
- Evitar hacer un uso erróneo de las actualizaciones de sensores y vistas, evitando así sus consecuentes efectos colaterales en el funcionamiento del simulador.

### 3.4. Agente

Nuestro agente extiende la clase `.^agent`" de JADE y gestiona el ciclo de vida del agente, incluyendo la configuración inicial, la toma de decisiones y el seguimiento de su entorno. Emplea un comportamiento periódico para evaluar continuamente su posición y actuar en consecuencia. Nuestro agente cuenta con una instancia del entorno, pues no puede utilizar el mapa de manera directa, ya que no debe conocer nada de él, a excepción de su posición y la de la casilla objetivo. Además, cuenta con un array de sensores, que suponen su memoria y un mapa que funciona como otra memoria, donde almacena el conocimiento que va teniendo conforme se mueve y ve el mapa original.

El agente utiliza tres comportamientos, los cuales se encuentran en el paquete `comportamientos`". Estos extienden de "Behaviour" de JADE y van destinados a actualizar la memoria, decidir el siguiente movimiento y hacer un recuento de los pasos totales para mostrarlos por pantalla al finalizar la simulación. En estos comportamientos, hemos utilizado una instancia del agente, desde la cual llamamos al método correspondiente que realiza cada una de estas funcionalidades, implementando manualmente los métodos `.^actionz` `"done"`de cada comportamiento para controlar qué se hace y cuándo debe hacerlo o dejar de hacerlo (cuando se ha alcanzado la casilla objetivo, básicamente).

La lógica relativa a la decisión de un nuevo movimiento se encuentra en la clase `.^star`", que realiza una búsqueda haciendo uso del conocido algoritmo A\*. La idea principal del funcionamiento de esta clase es que contiene una heurística para estimar la distancia del origen al destino (posición del agente a posición de la casilla objetivo), que es usada en el método `"busqueda"`, el cual es el encargado de dar un camino en un mapa dado desde una posición destino a una origen con el ya comentado algoritmo A\*.

### 3.5. Diagrama de clases UML con el diseño del simulador

El proceso de diseño e implementación de esta práctica ha sido llevado a cabo mediante el refinamiento sucesivo. Hemos usado un diagrama UML de clases para diseñar y representar la

arquitectura del software de nuestro simulador, el cual ha ido sufriendo cambios motivados por errores que hemos ido detectando, junto con el feedback recibido por el profesor en las sesiones de prácticas. Como nos parece interesante mostrar la evolución de nuestro software a lo largo del proceso, pasamos a mostrar los diferentes estados (versiones) que ha tenido nuestro diseño:

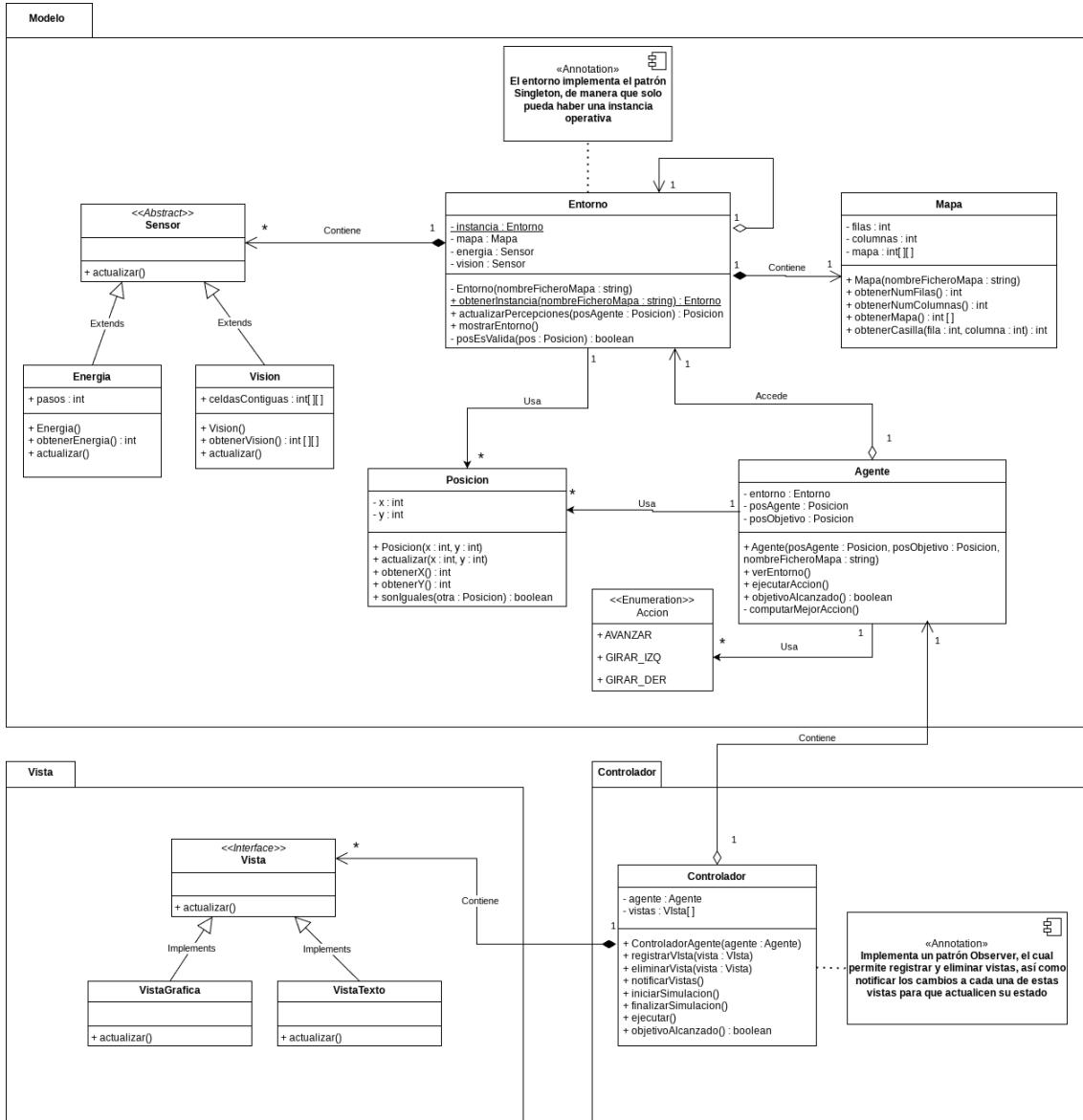


Figura 1: Versión 1 del diagrama de clases UML

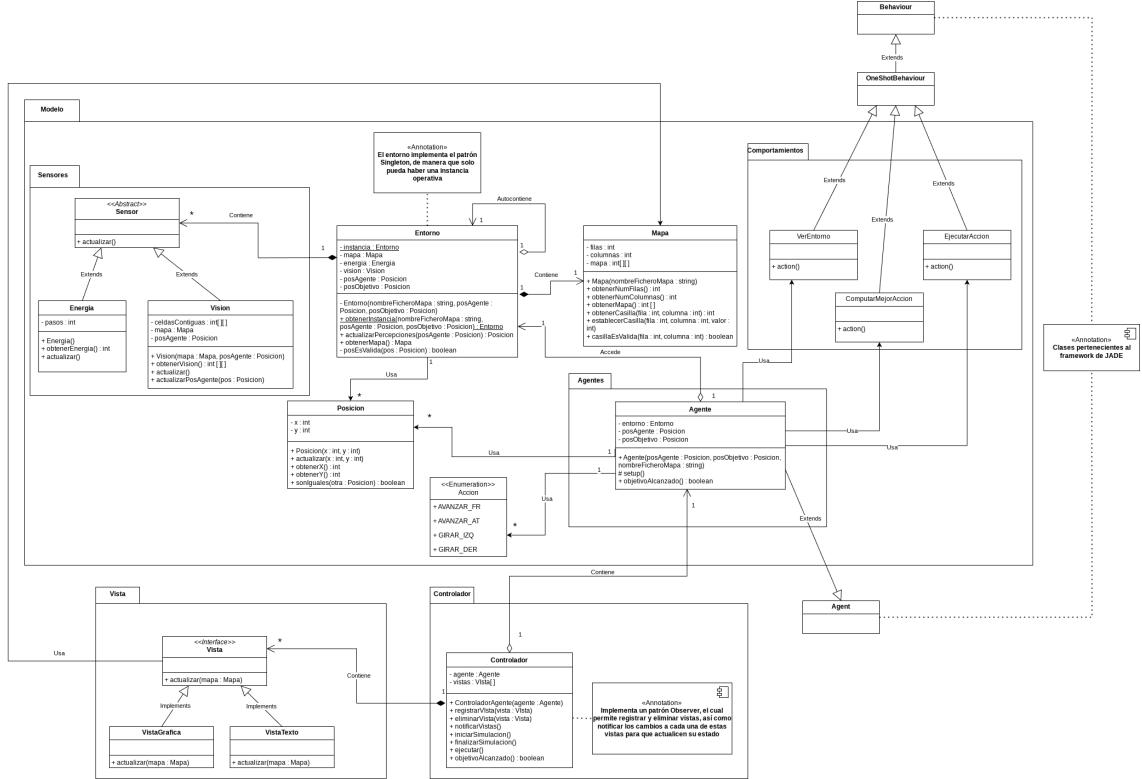


Figura 2: Versión 2 del diagrama de clases UML

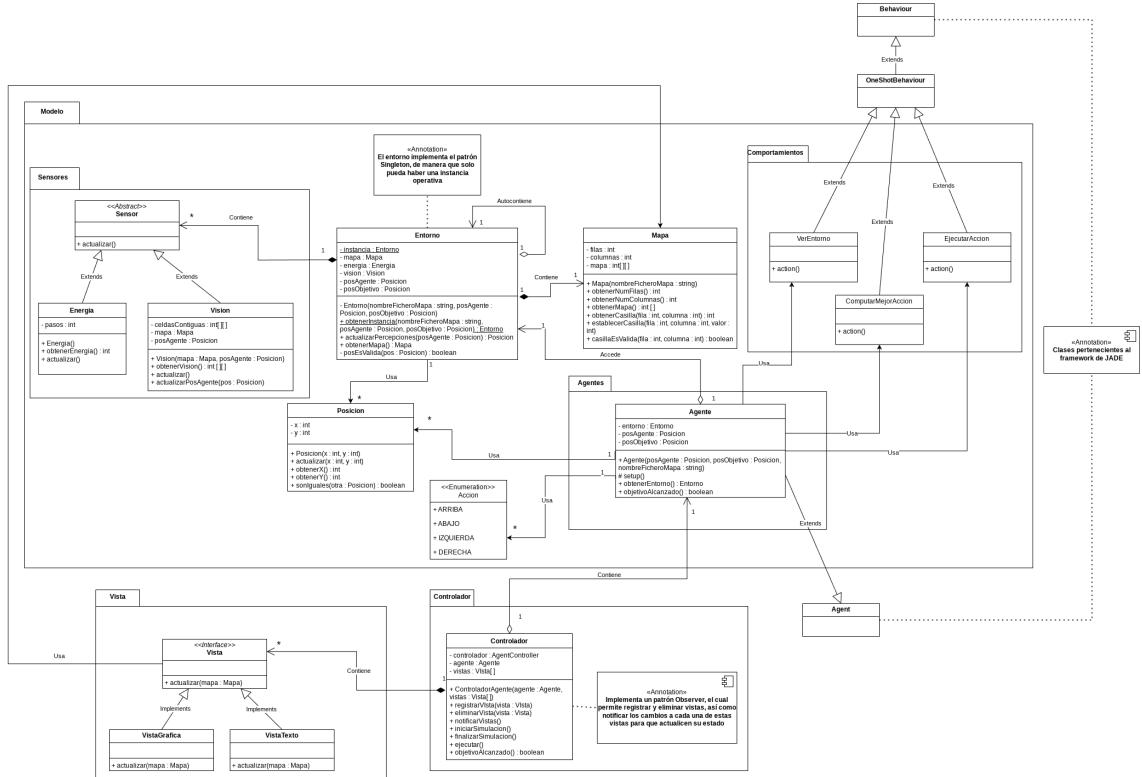


Figura 3: Versión 3 del diagrama de clases UML

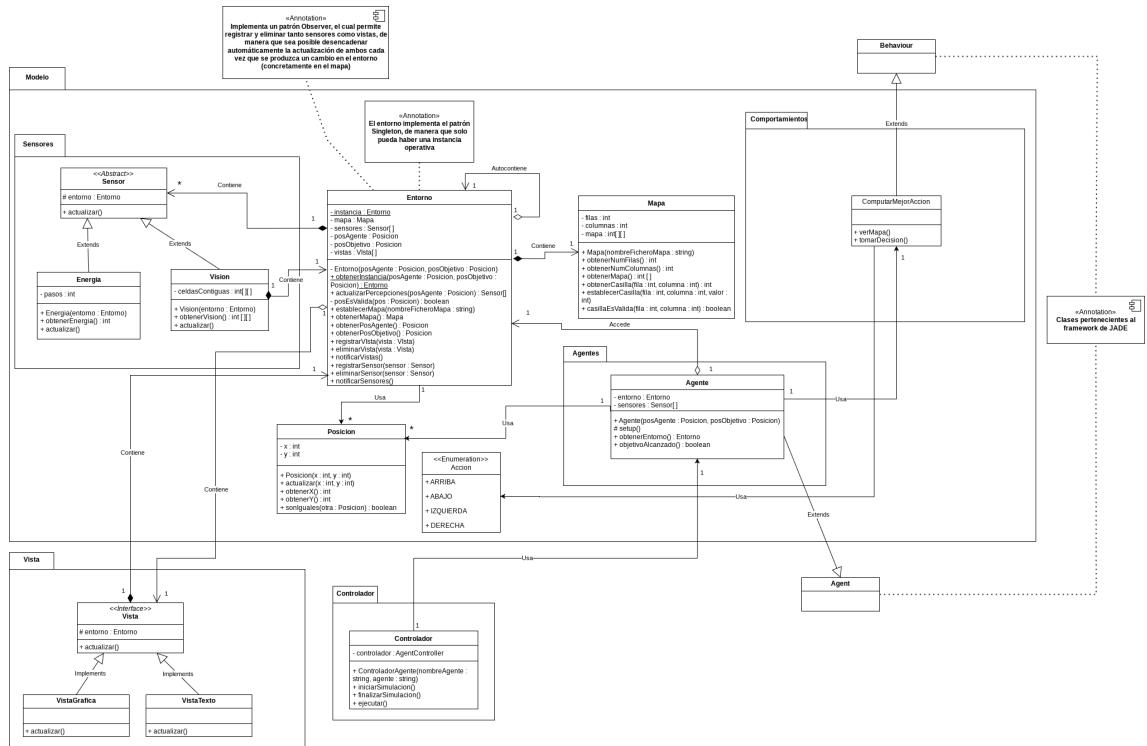


Figura 4: Versión 4 del diagrama de clases UML

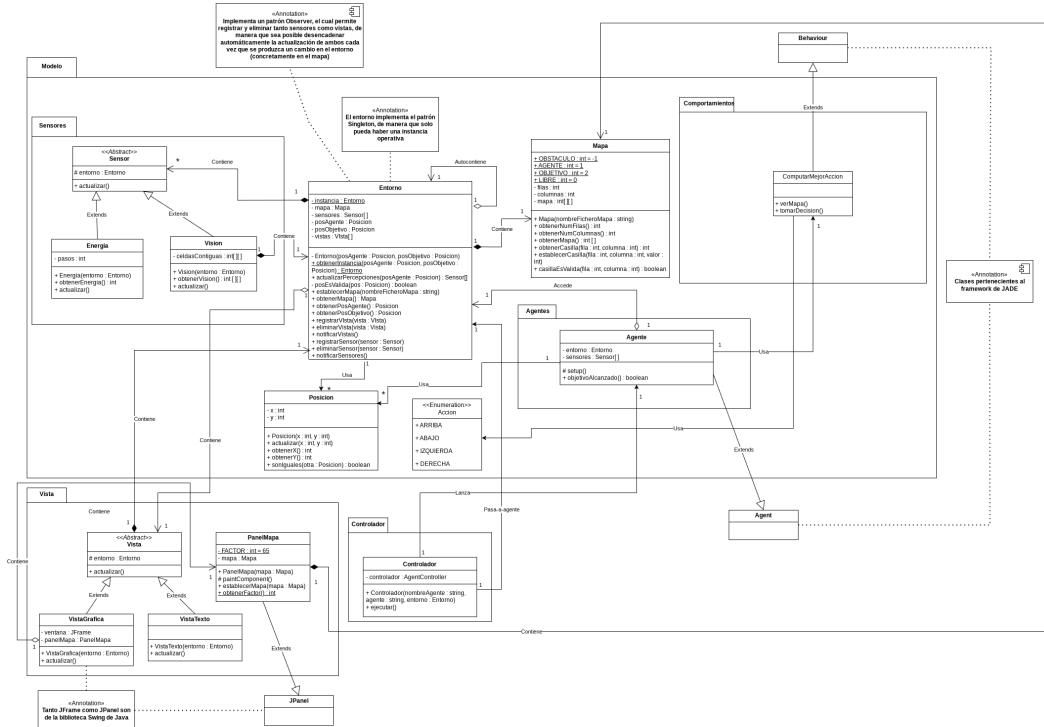


Figura 5: Versión 5 del diagrama de clases UML

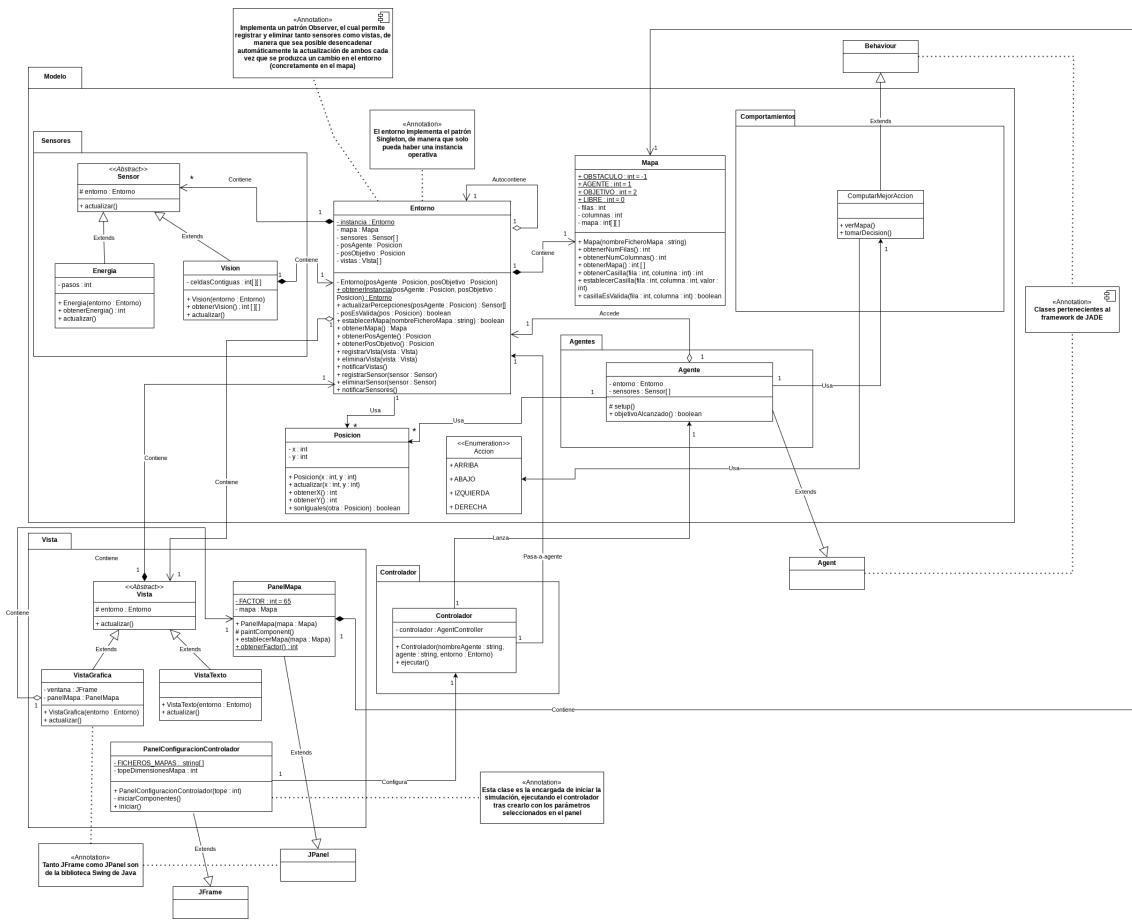


Figura 6: Versión 6 del diagrama de clases UML

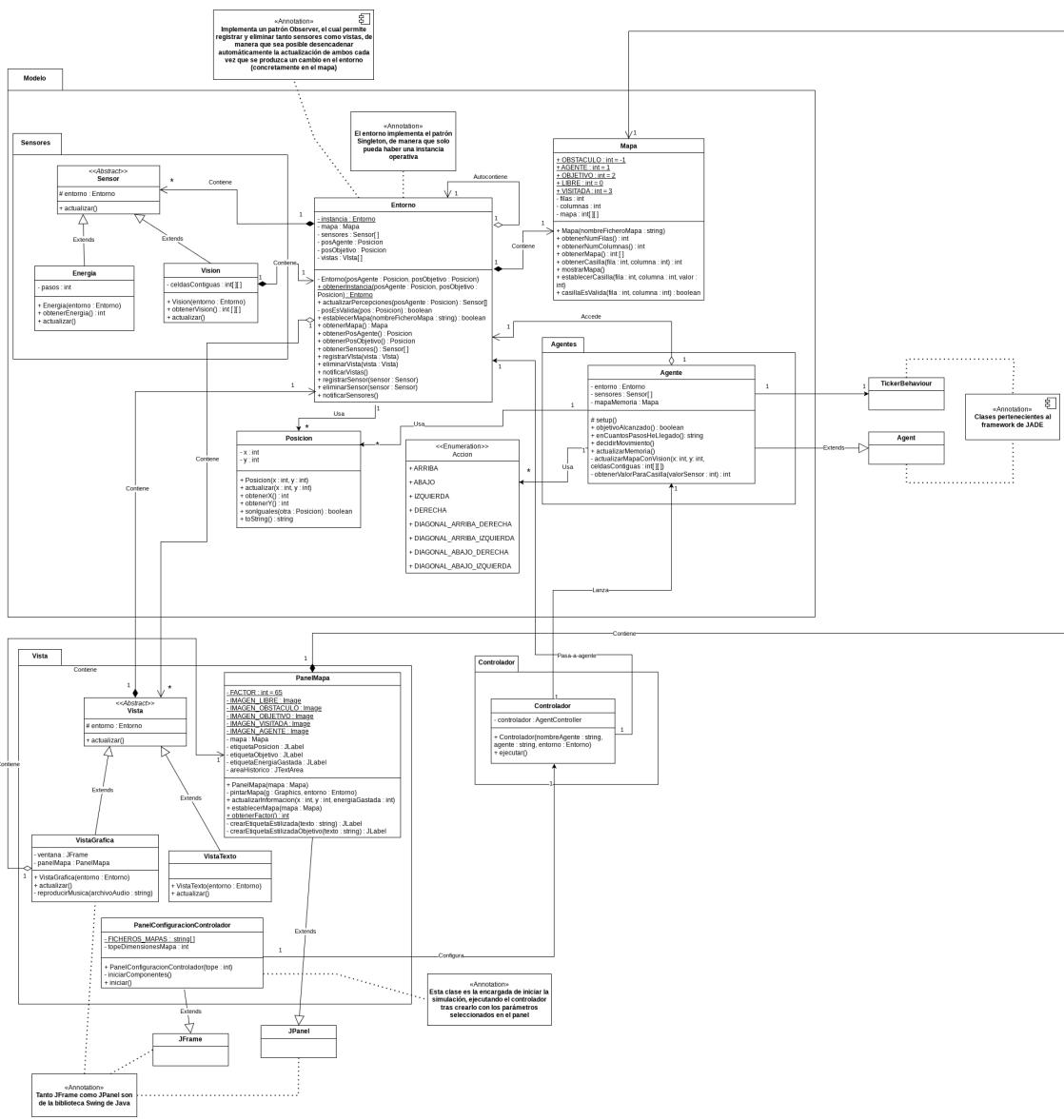


Figura 7: Versión final previa del diagrama de clases UML

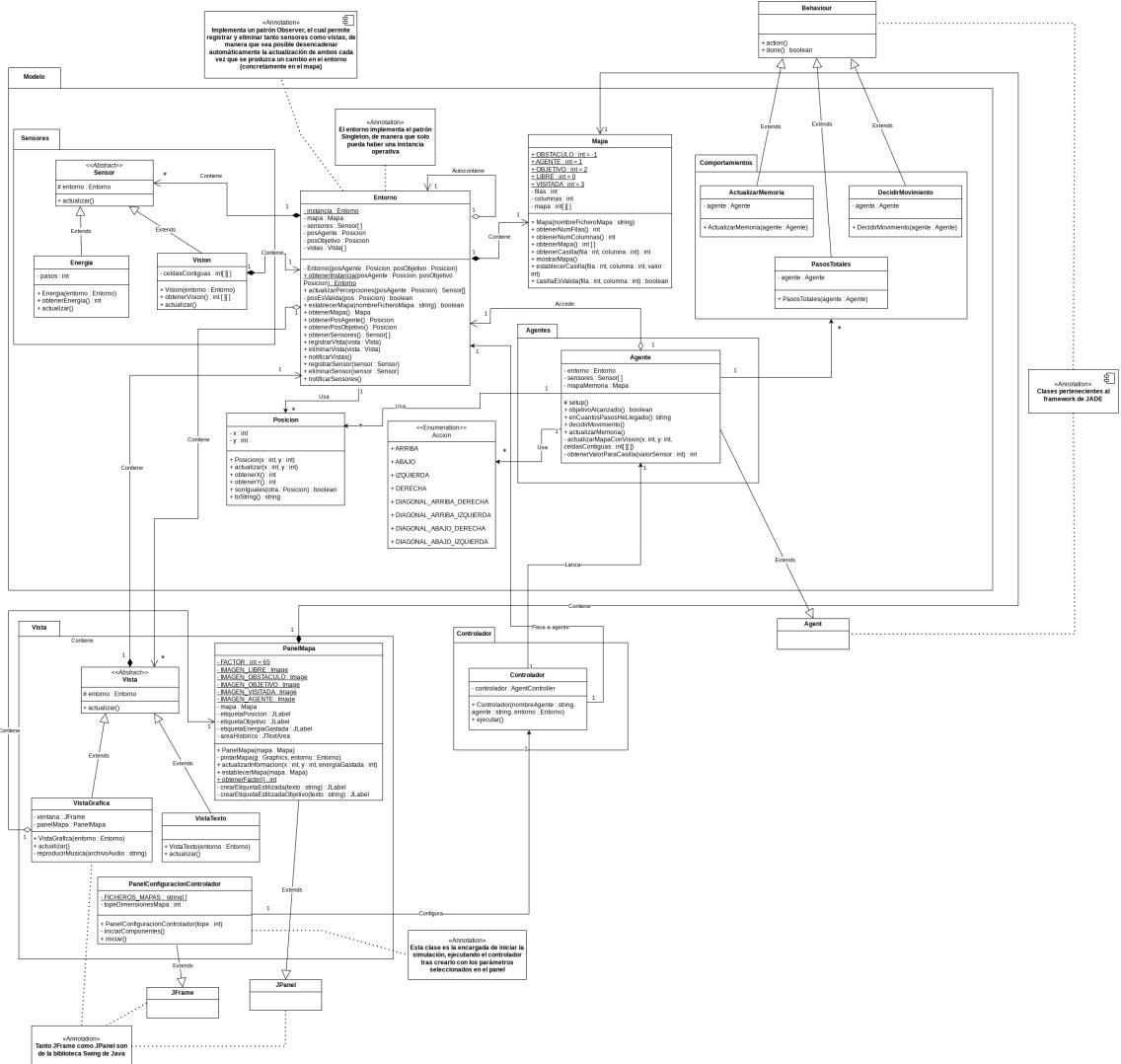


Figura 8: Versión final del diagrama de clases UML

## 4. Dificultades encontradas

En primer lugar, nos costó asimilar las limitaciones que tenía que tener el agente con respecto al conocimiento del entorno, pues en un principio contemplábamos que este podía tener cierto conocimiento, como, por ejemplo, la dimensión del mapa, de cara a establecer la memoria atendiendo a este dato. Esto nos llevó a diseñar la arquitectura de una manera algo errónea, pues el agente usaba el propio mapa, aunque fuera a través del entorno.

En segundo lugar, tuvimos algunas dificultades para incorporar el patrón Observer a la arquitectura, pues en un principio lo diseñamos para que se aplicara al Controlador, de manera que este fuera el objeto observado y solamente por las vistas, lo que no era la mejor opción, pues los cambios sobre el mapa son responsabilidad del entorno y, por ende, este último es objeto que debía ser observado, y no solo por las vistas, sino también por los sensores.

En tercer lugar y último lugar, también encontramos algunas dificultades a la hora de decidir el diseño del agente, pues nos centramos demasiado en los tipos de comportamientos del framework de JADE que debíamos utilizar para que el agente fuera consistente con la metodología de este framework. Despues de pensarla y debatirla, junto con algunos comentarios del profesor, optamos por simplificar esta parte y usar simplemente un comportamiento periódico para ir comprobando si se cumple la condición de parada de la simulación que, en este caso, es la posición del agente es la misma que la posición de la casilla objetivo. Mientras no se cumple, se actualizan los sensores y se decide la siguiente acción a llevar a cabo en el mapa. Cabe reseñar que en lugar de buscar

comportamientos específicos del framework que fueran lo suficientemente semánticos, optamos por encapsular esos comportamientos en forma de métodos del agente y de otra clase que se utiliza para aplicar el algoritmos de búsqueda A\*.

## 5. Pruebas y resultados

A continuación, vamos a ilustrar el funcionamiento final del simulador, mostrando algunas pruebas realizadas, haciendo uso de los mapas proporcionados por el profesor:

### 5.1. Prueba 1: mapa sin obstáculos

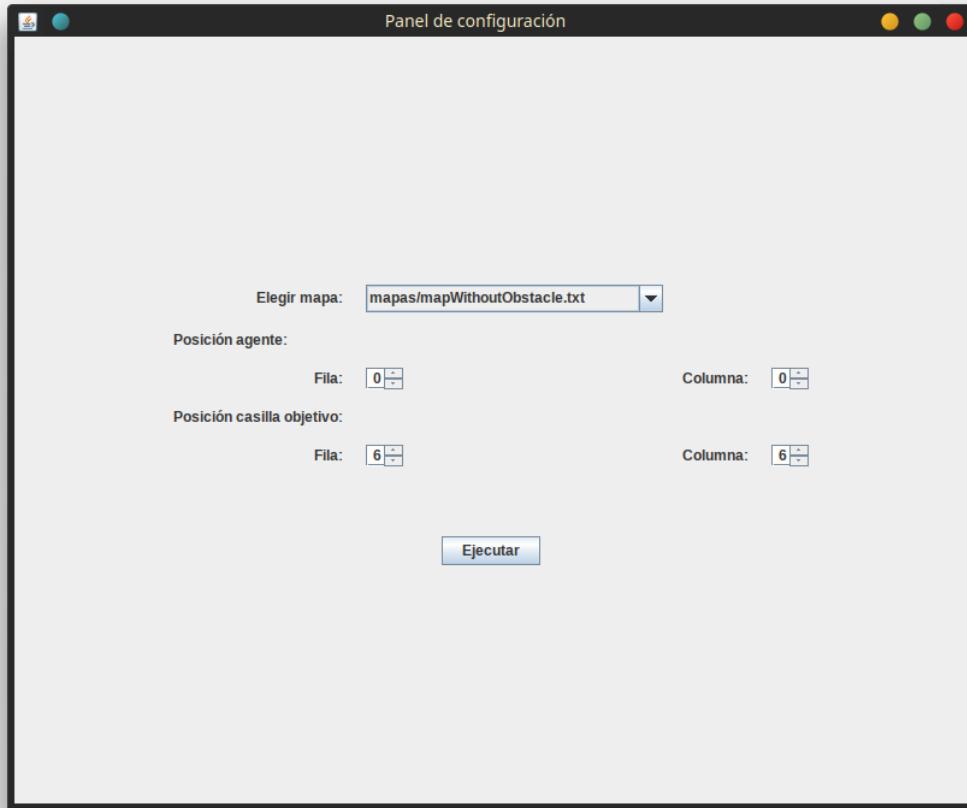


Figura 9: Configuración de la prueba 1

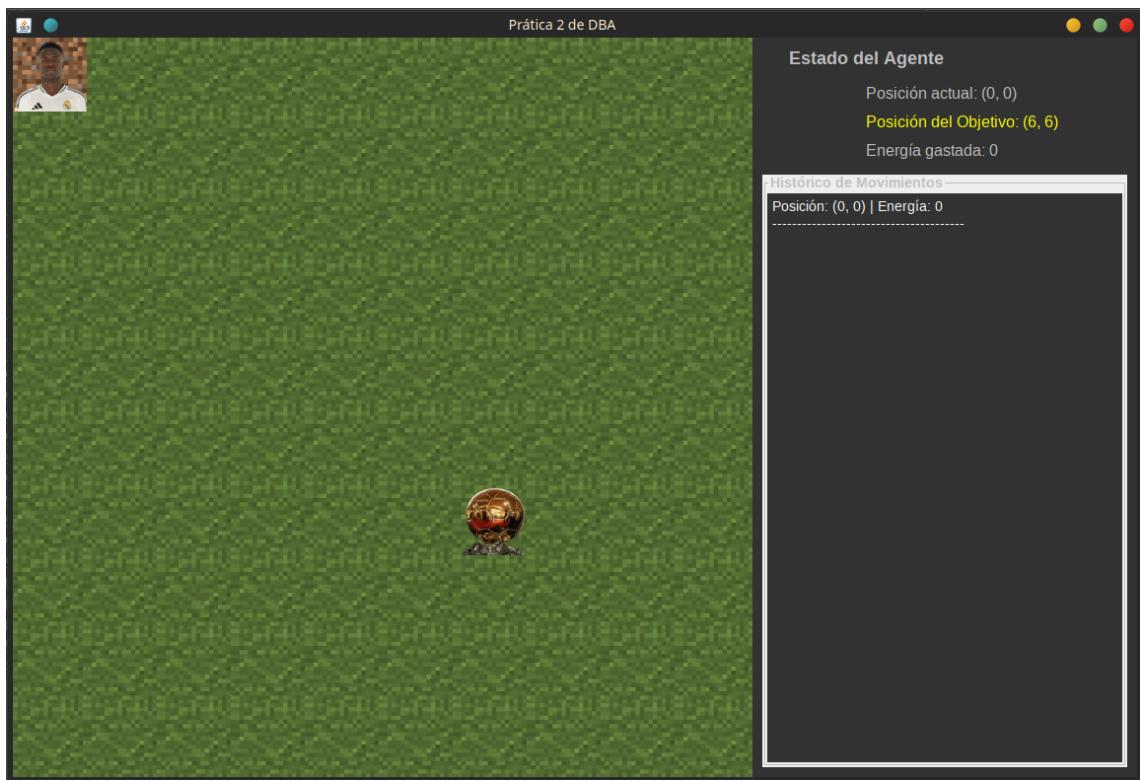


Figura 10: Estado inicial de la prueba 1

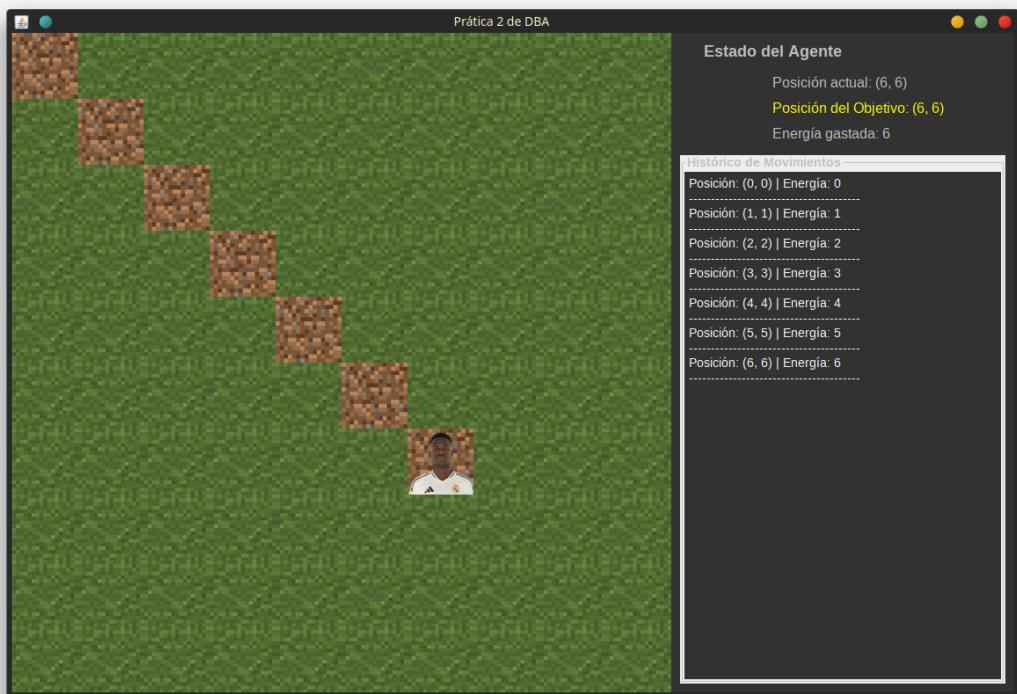


Figura 11: Resultado de la prueba 1

## 5.2. Prueba 2: mapa con muro horizontal

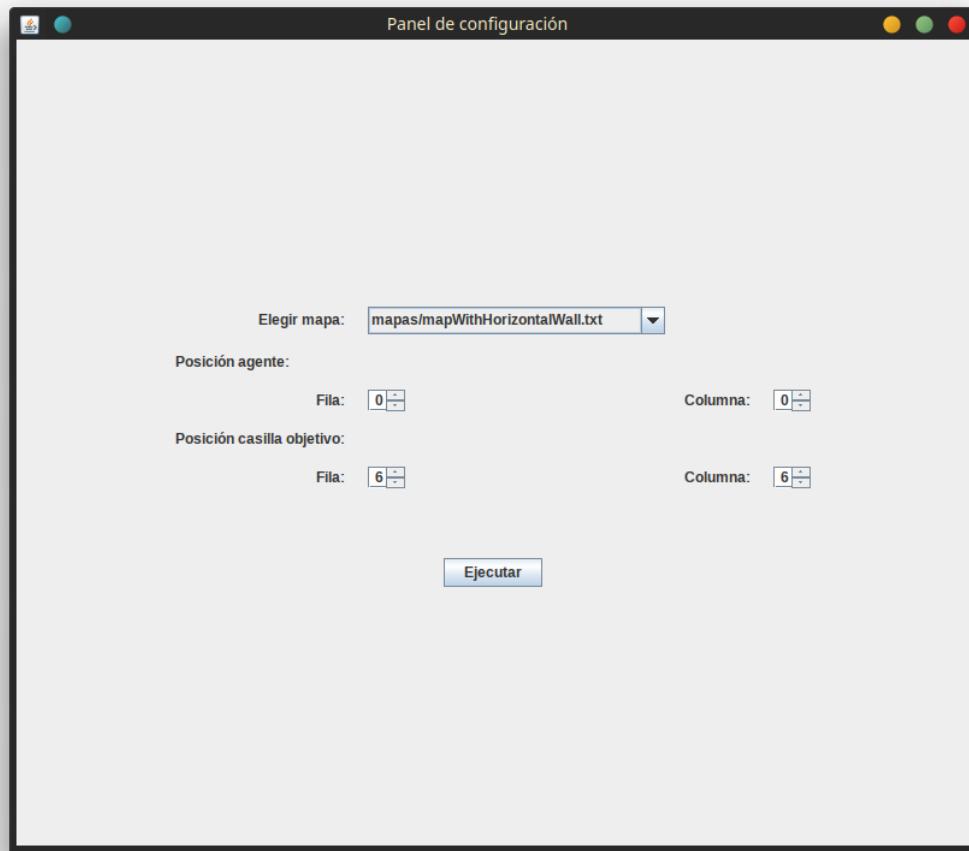


Figura 12: Configuración de la prueba 2

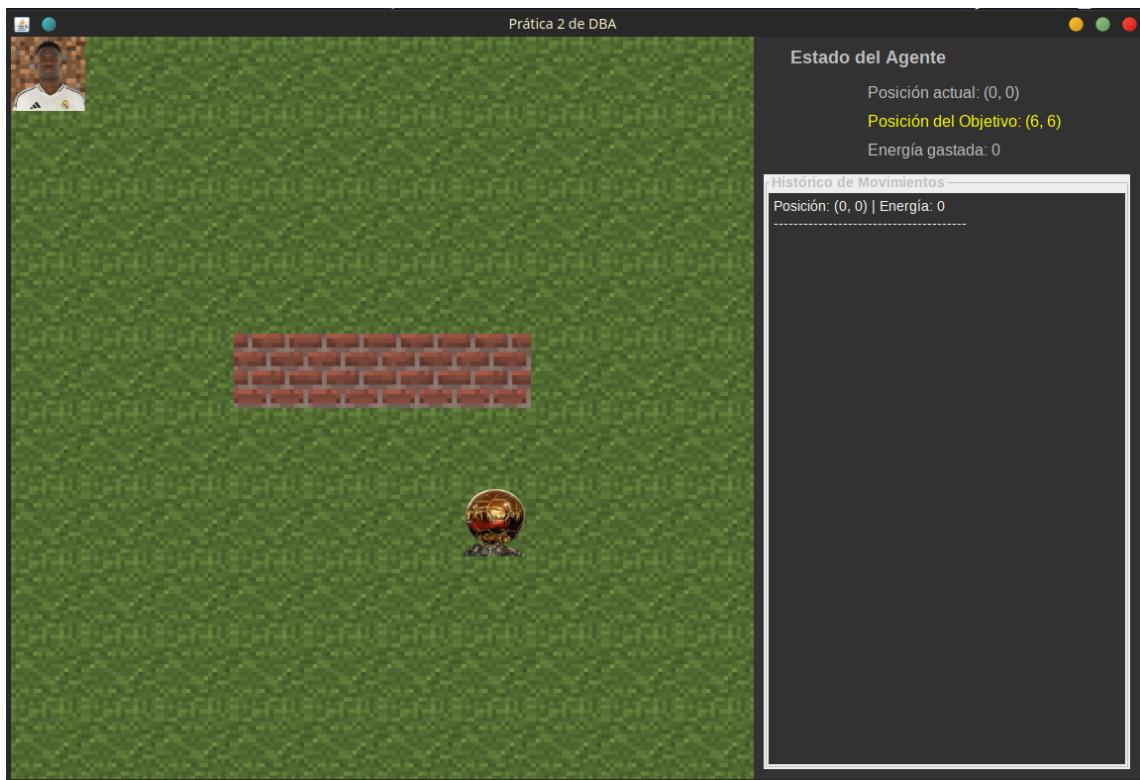


Figura 13: Estado inicial de la prueba 2

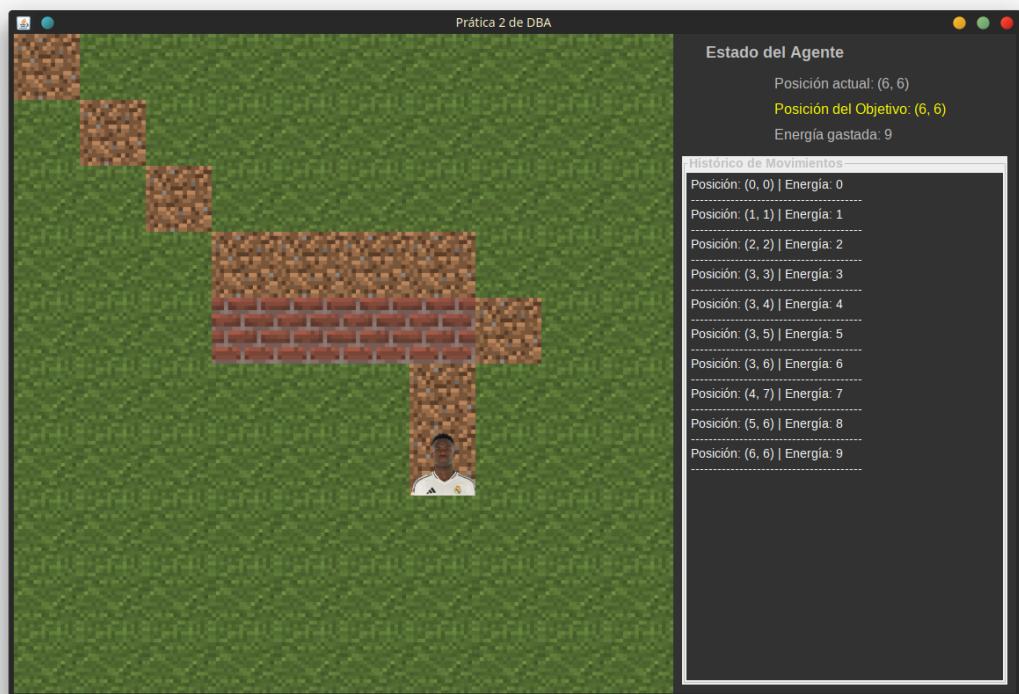


Figura 14: Resultado de la prueba 2

### 5.3. Prueba 3: mapa con muro vertical

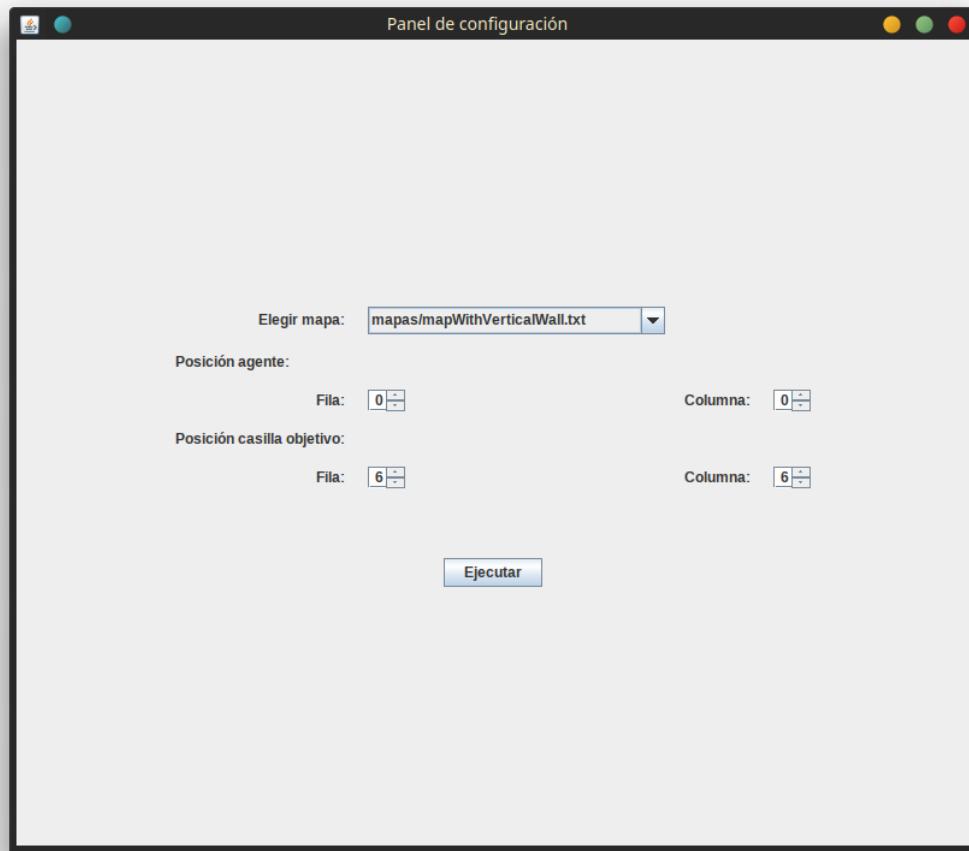


Figura 15: Configuración de la prueba 3

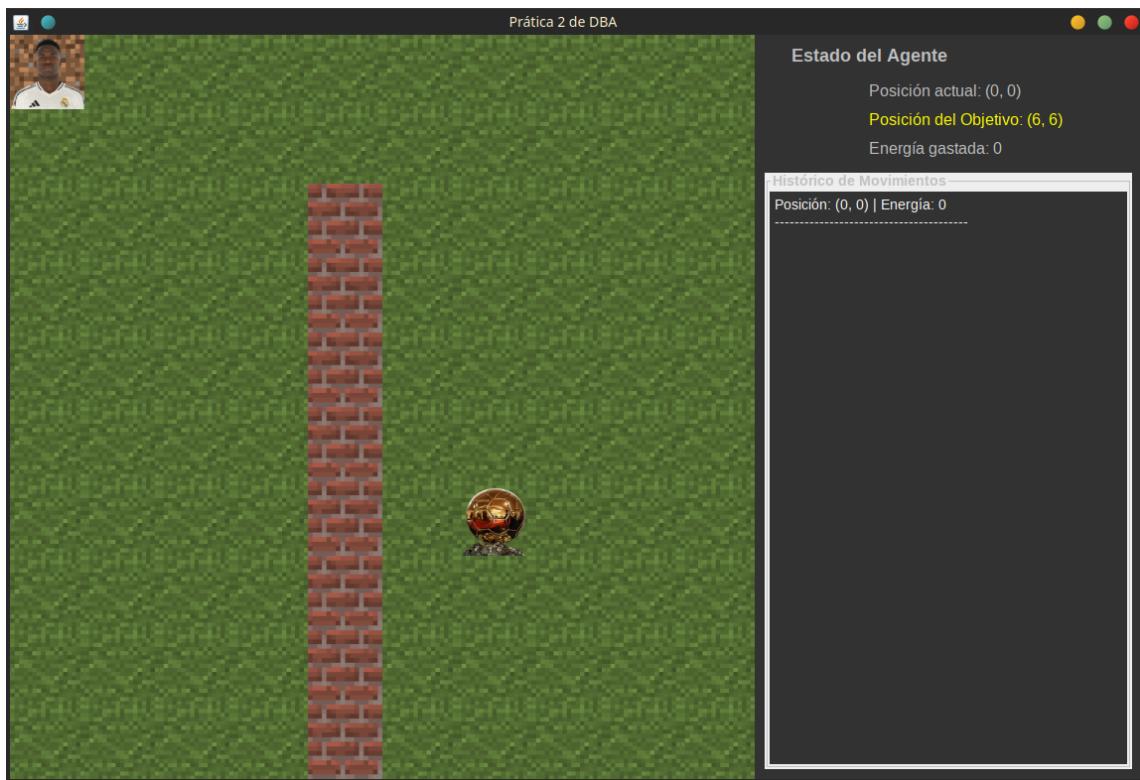


Figura 16: Estado inicial de la prueba 3



Figura 17: Resultado de la prueba 3

#### 5.4. Prueba 4: mapa con muro diagonal

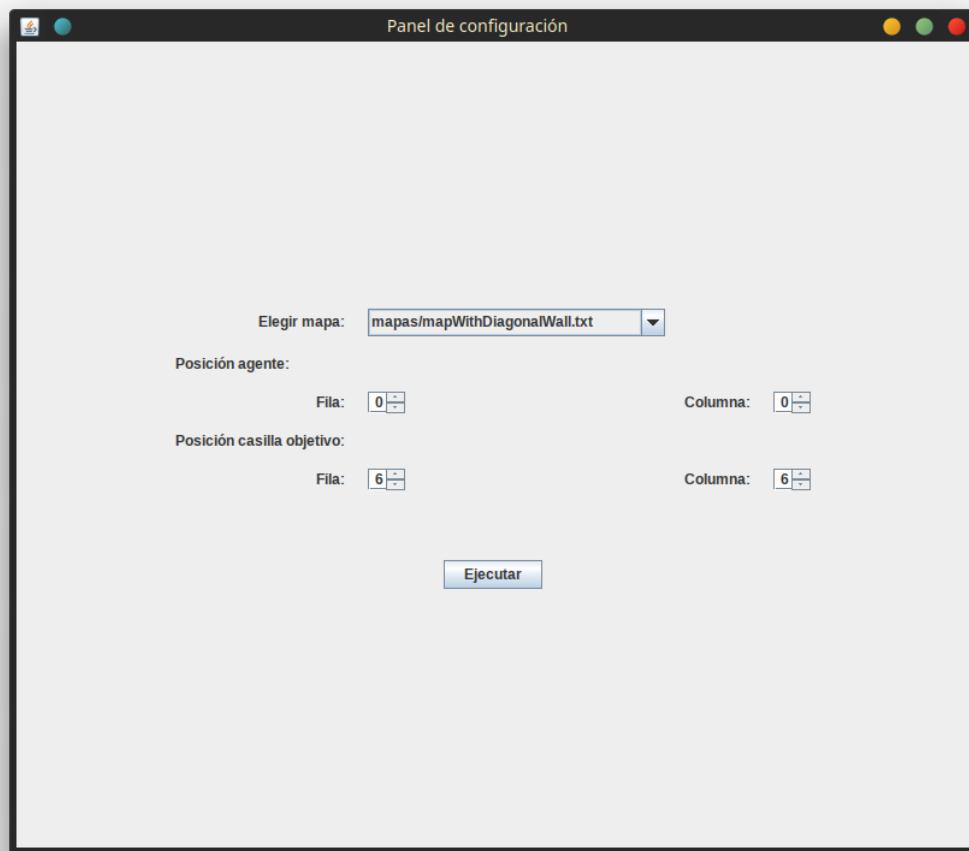


Figura 18: Configuración de la prueba 4

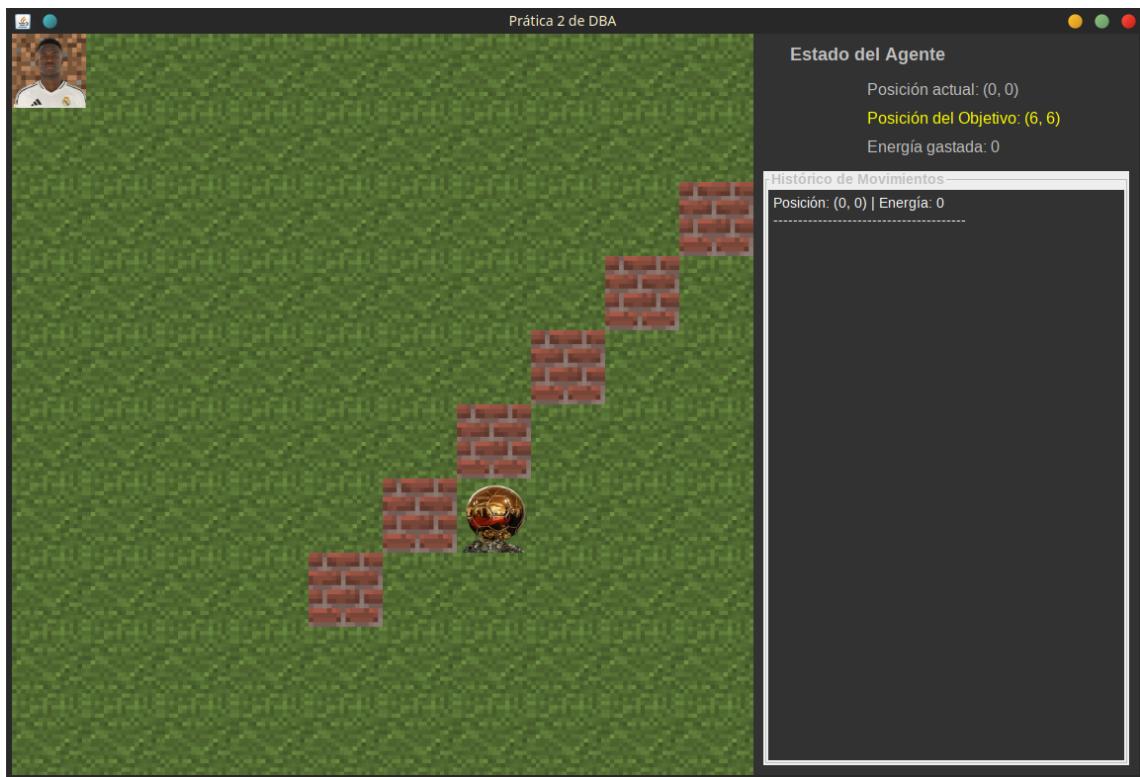


Figura 19: Estado inicial de la prueba 4

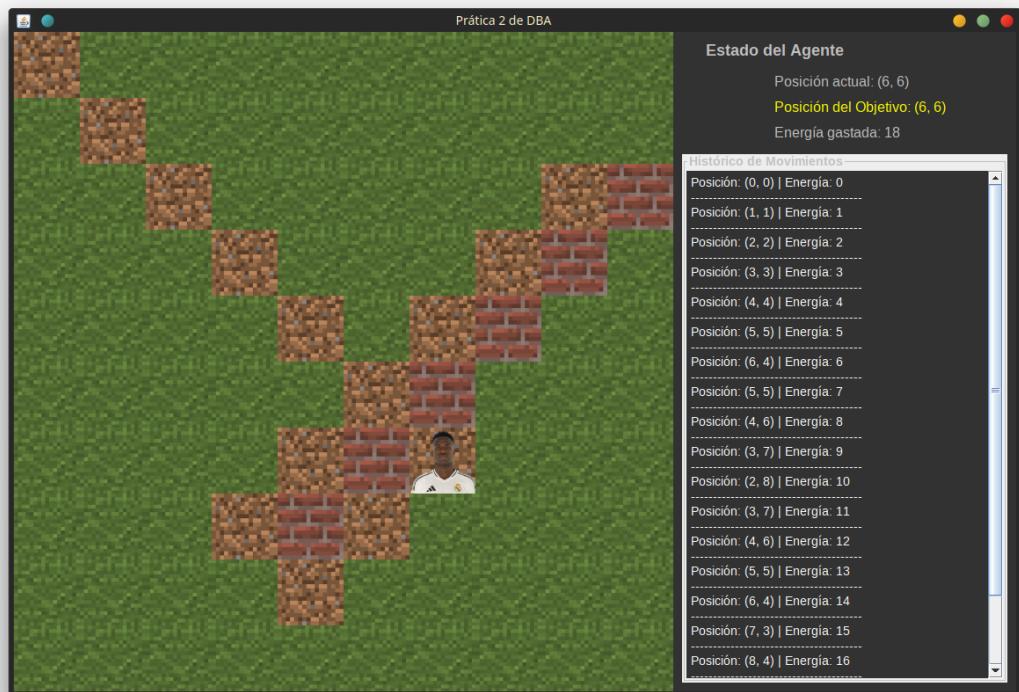


Figura 20: Resultado de la prueba 4

## 5.5. Prueba 5: mapa con muro convexo

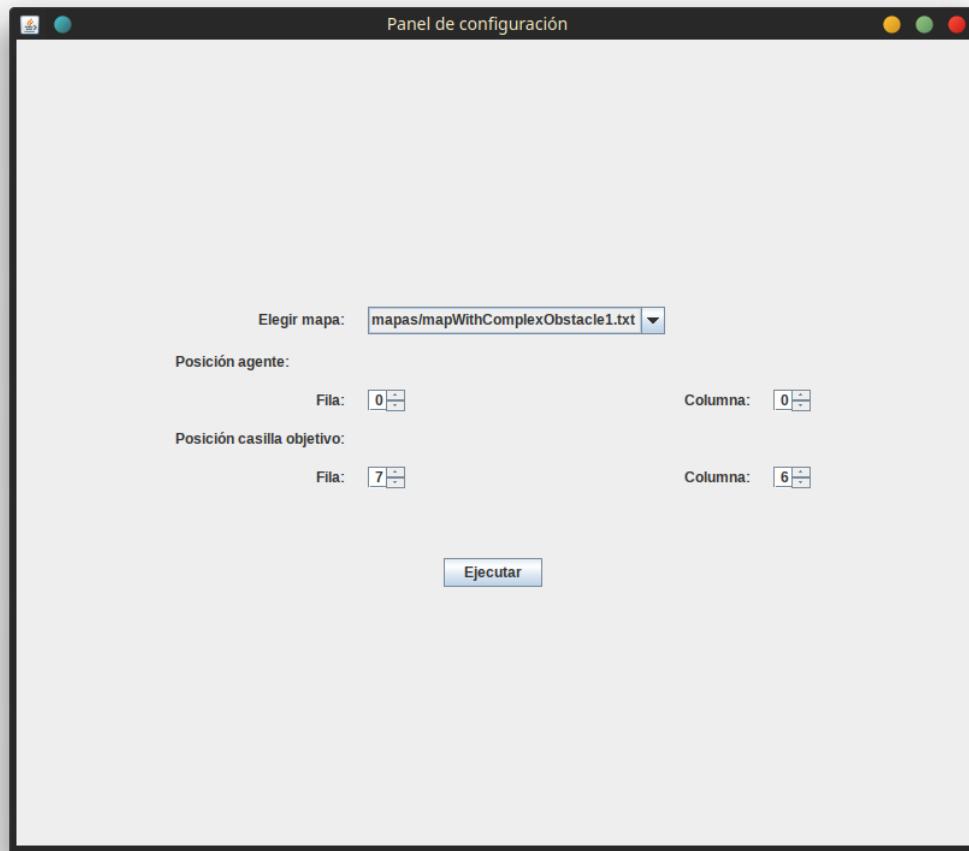


Figura 21: Configuración de la prueba 5



Figura 22: Estado inicial de la prueba 5

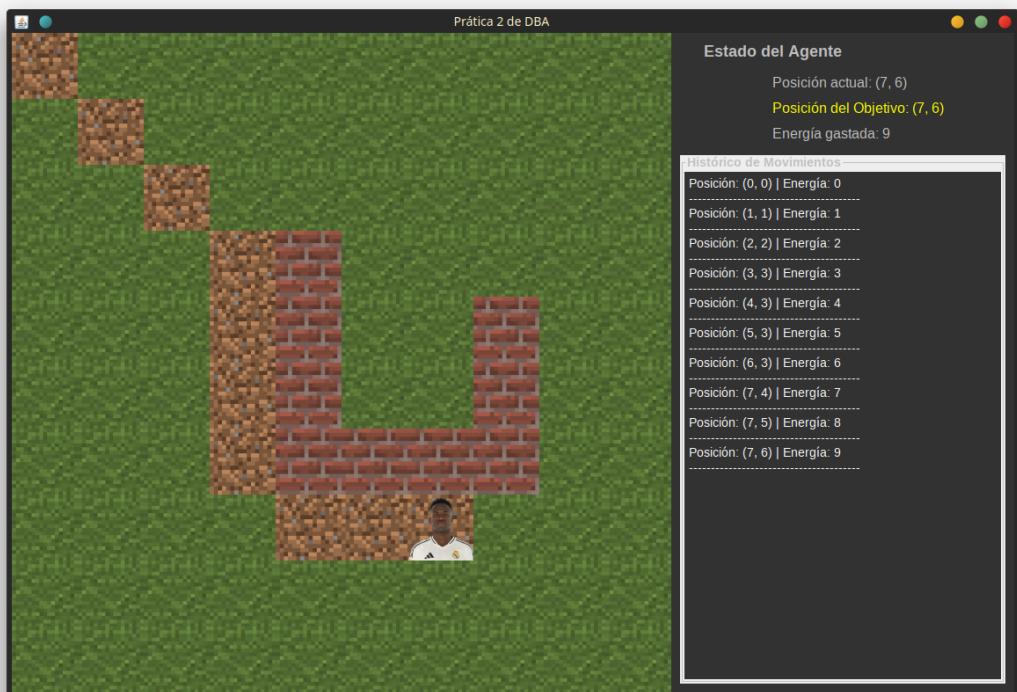


Figura 23: Resultado de la prueba 5

## 5.6. Prueba 6: mapa con muro cóncavo

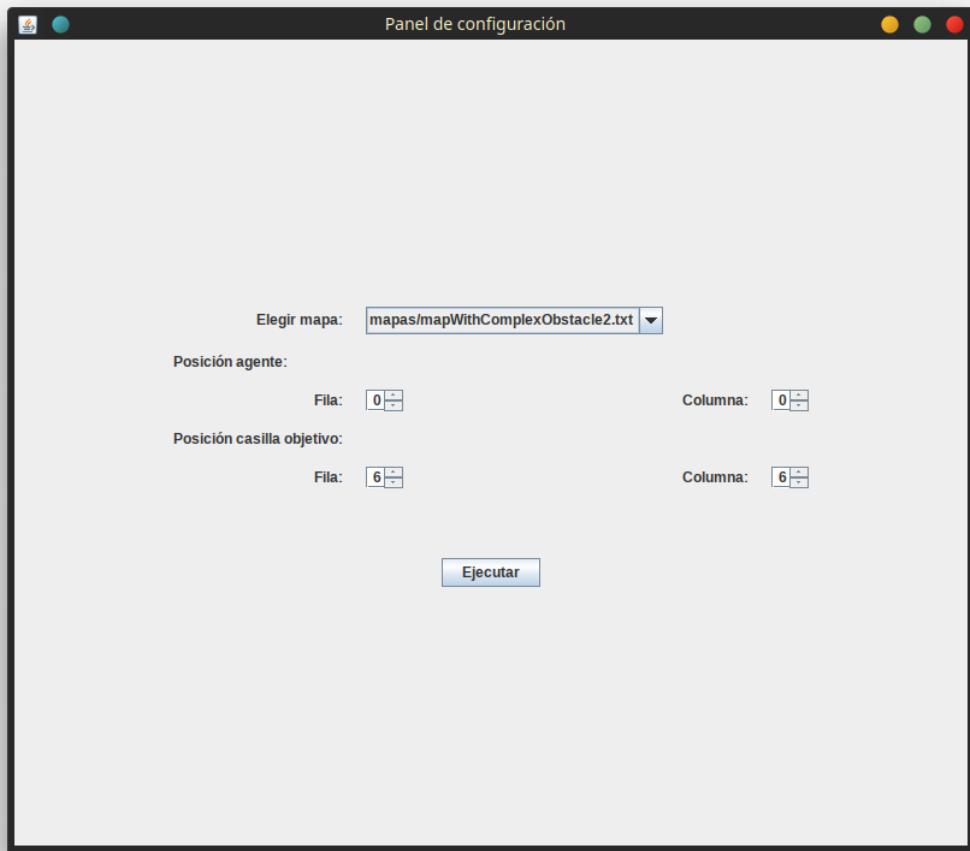


Figura 24: Configuración de la prueba 6

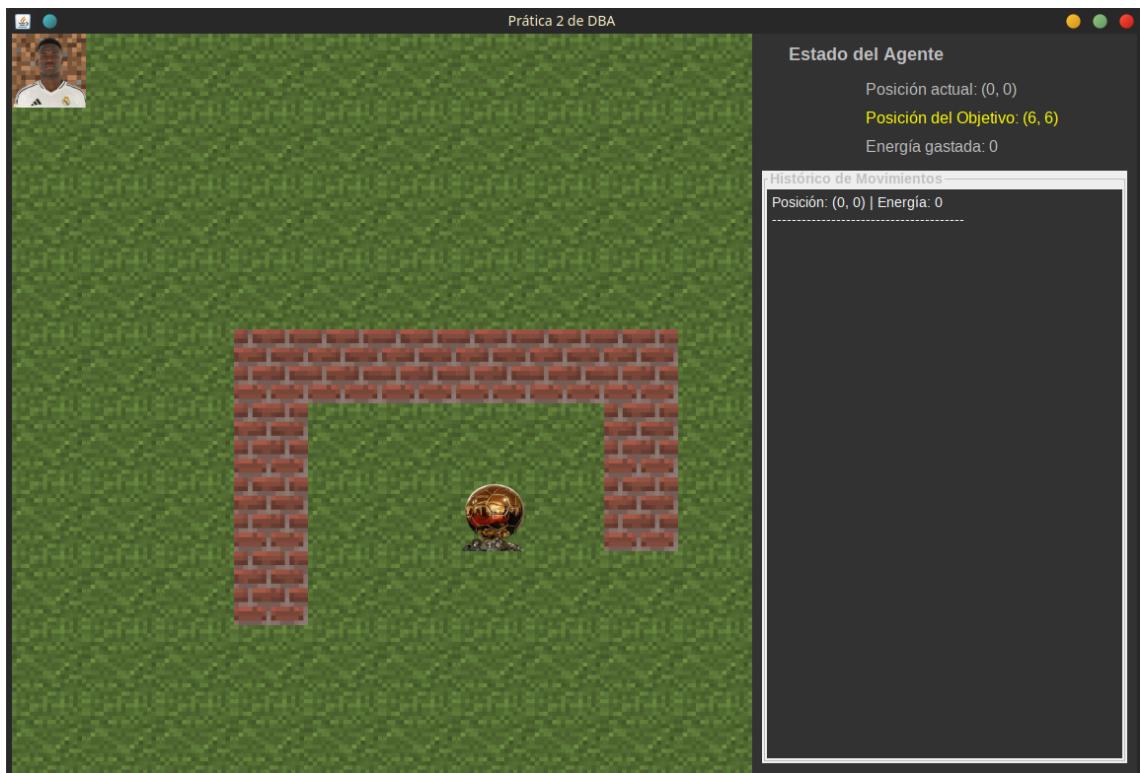


Figura 25: Estado inicial de la prueba 6

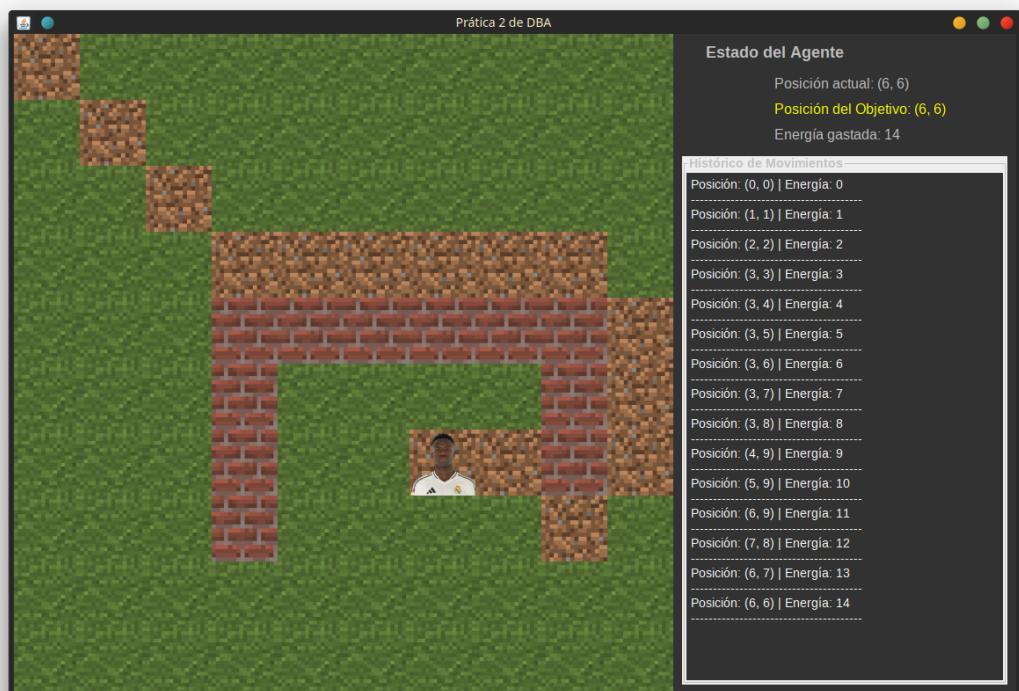


Figura 26: Resultado de la prueba 6

## 5.7. Prueba 7: mapa con muro más complejo

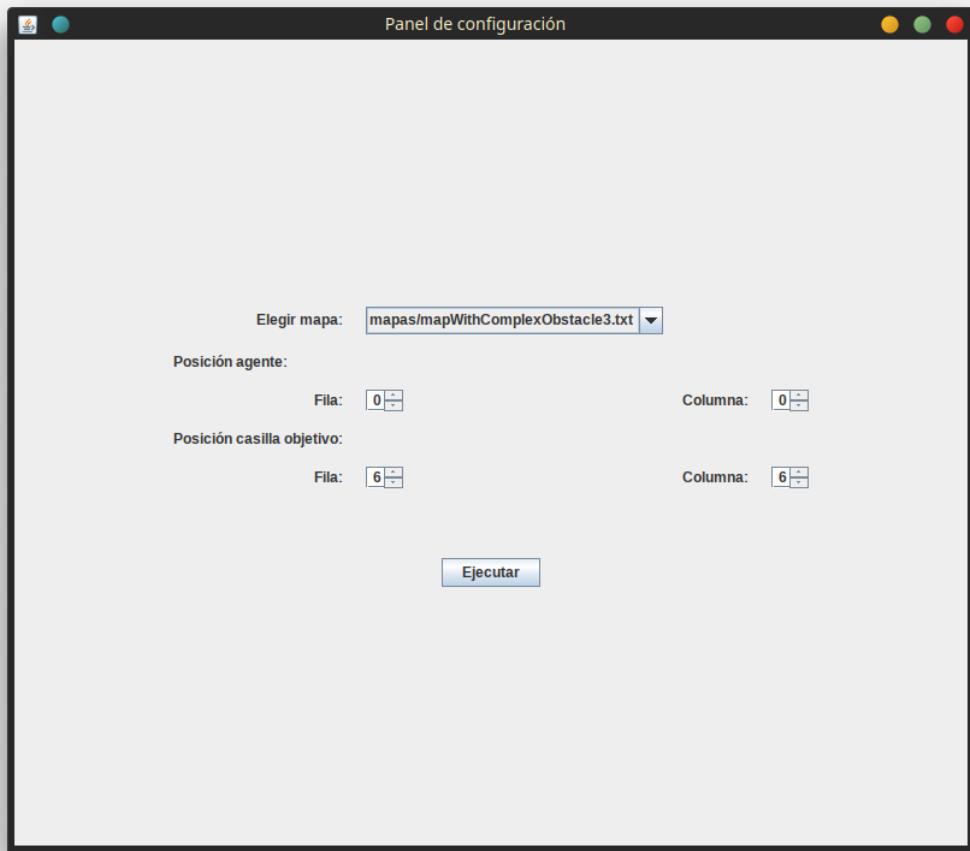


Figura 27: Configuración de la prueba 7

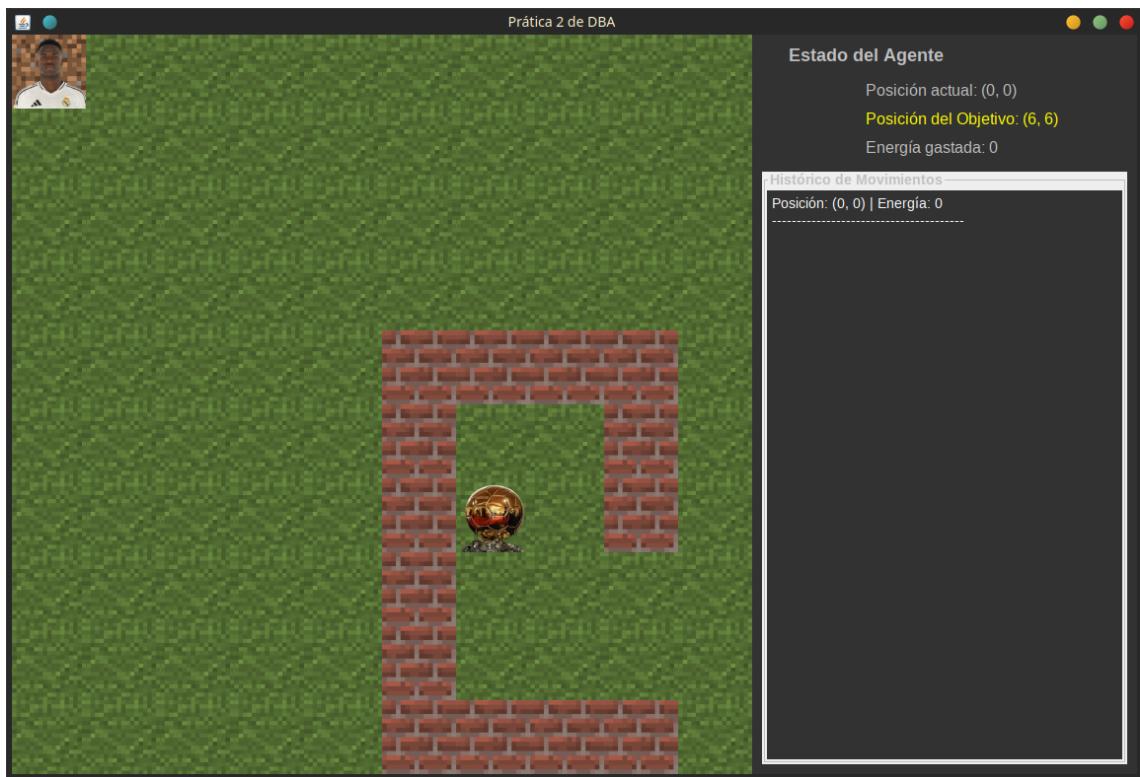


Figura 28: Estado inicial de la prueba 7

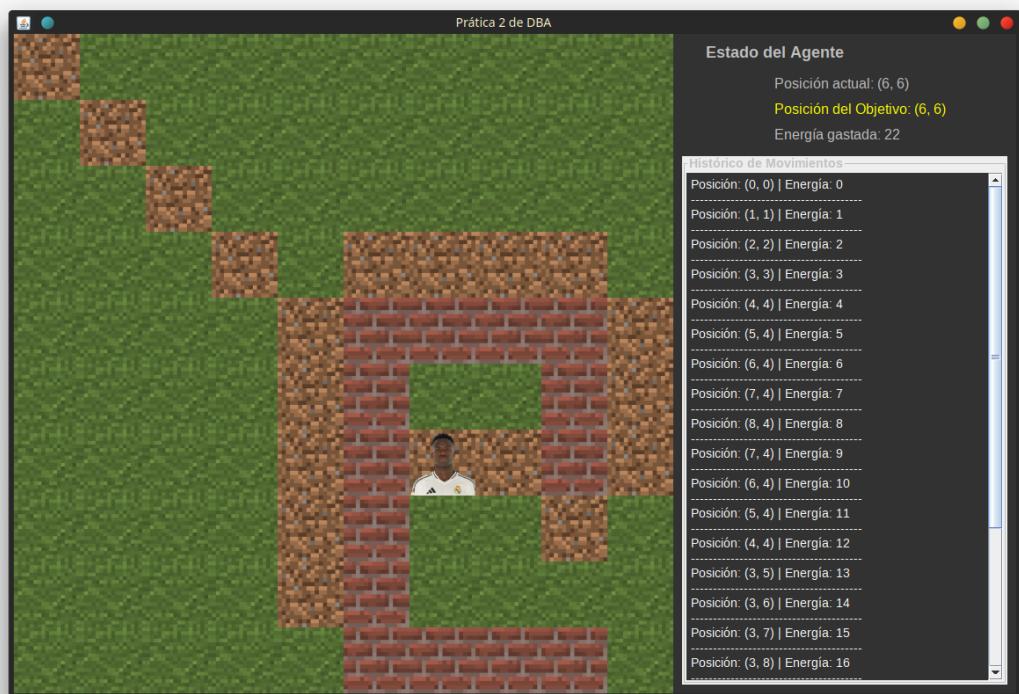


Figura 29: Resultado de la prueba 7

## 5.8. Prueba 8: mapa personalizado con dificultad extra

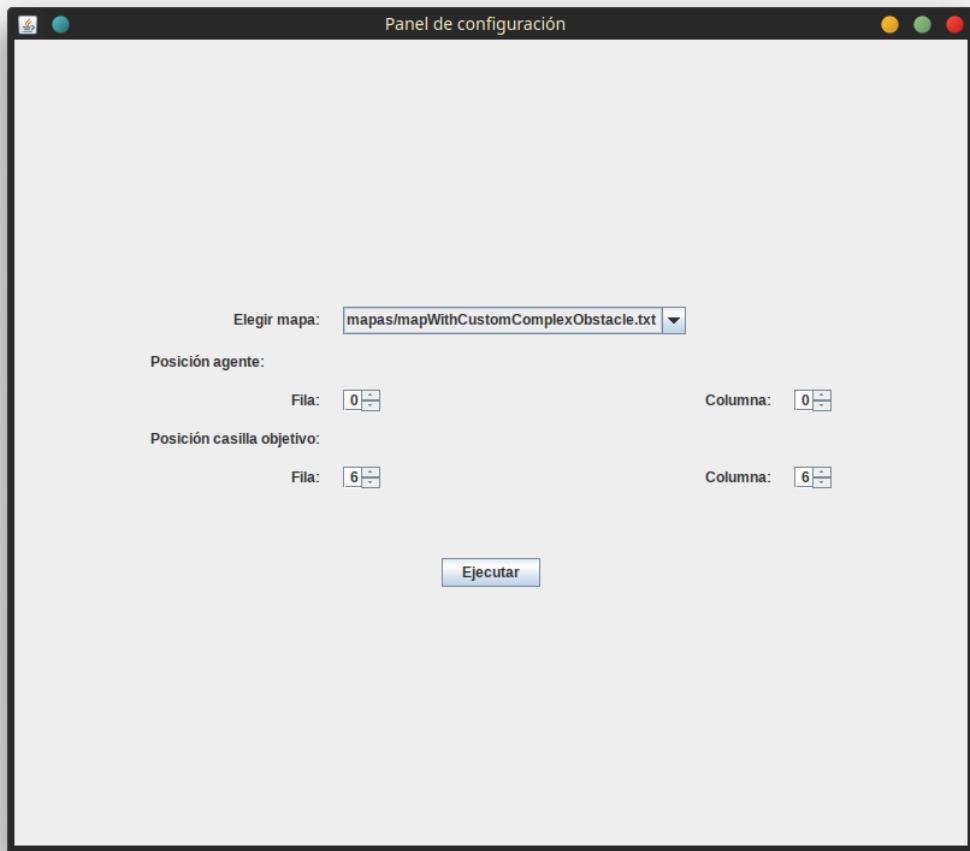


Figura 30: Configuración de la prueba 8

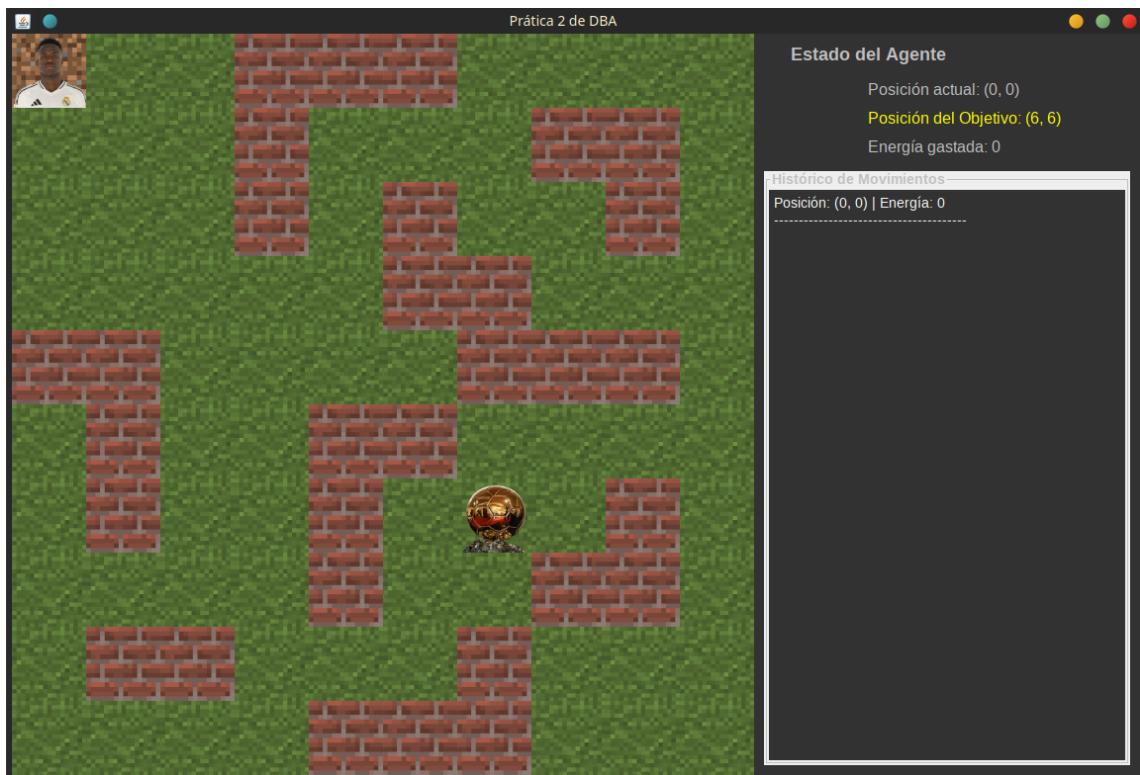


Figura 31: Estado inicial de la prueba 8

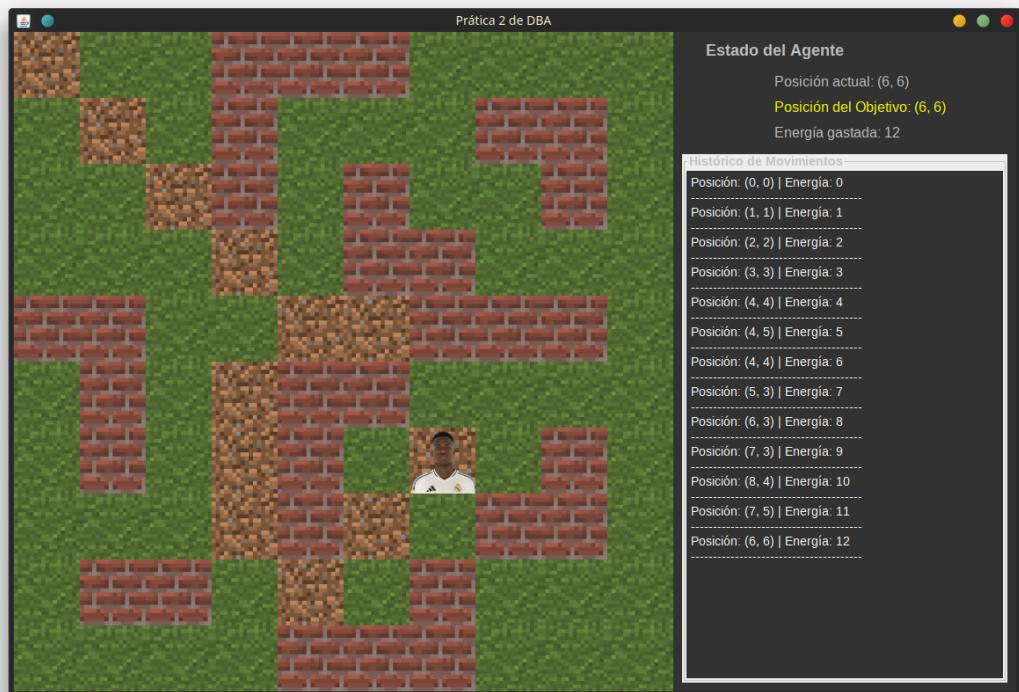


Figura 32: Resultado de la prueba 8

## 6. Instrucciones de funcionamiento del simulador

Una vez se ejecuta el simulador, aparece un panel de configuración con los datos básicos iniciales para lanzar al agente, los cuales son el mapa, la posición inicial del agente y la posición de la casilla objetivo. Una vez que se han puesto, basta con pulsar sobre el botón “Ejecutar” de la parte inferior del panel. Tras esto, se inicia la simulación, la cual termina en el momento en que el agente se sitúa sobre la casilla objetivo.

En el caso en el que se desee añadir un nuevo mapa al simulador, hay que seguir los siguientes pasos:

- Guardar el mapa con la estructura correcta como un fichero de texto plano en el directorio “mapas” del proyecto.
- Escribir la ruta relativa desde el directorio “mapas” del proyecto al final de la constante “FICHEROS\_MAPAS” de la clase “PanelConfiguracionControlador”, para añadir, de esa manera, la cadena al final de ese vector de cadenas, el cual muestra estas cadenas como opciones a la hora de elegir el mapa en el panel de configuración inicial que aparece al ejecutar el proyecto.

## 7. Conclusiones

El simulador implementado ha logrado ser funcional, cumpliendo con los requisitos planteados, como la gestión interna de estados del agente y evitar obstáculos de diferentes tipos, consumiendo, además, la mínima cantidad de energía (pasos) posible.

El desarrollo de esta práctica nos ha sido satisfactorio, pues hemos disfrutado del proceso, junto con el resultado final. Eso sí, hay algunas lecciones que nos hemos llevado, pues nos hemos dado cuenta de que muchos aspectos del diseño e implementación del software los llevamos aprendidos de una forma teórica y poco flexible. Hemos visto claro que es vital conseguir adquirir la intuición y la destreza suficientes como para concebir los patrones de diseño de una manera más abierta, con una mentalidad más extendida y creativa.

Obviamente, hay aspectos que mejorar, tanto de diseño como de implementación, dado que hemos desarrollado la práctica entre cinco integrantes en un tiempo limitado, teniendo que lidiar también con aspectos relativos a la coordinación y gestión colectiva del proyecto.

Para finalizar, queremos reseñar que creemos que esta práctica ha contribuido positivamente a nuestra formación, pues ha implicado lidiar con aspectos técnicos y de trabajo en equipo, pero no de una manera aséptica, sino disfrutando del proceso, dado que la práctica ha resultado atractiva de abordar y motivante a la hora de ir viendo las mejoras y los resultados.