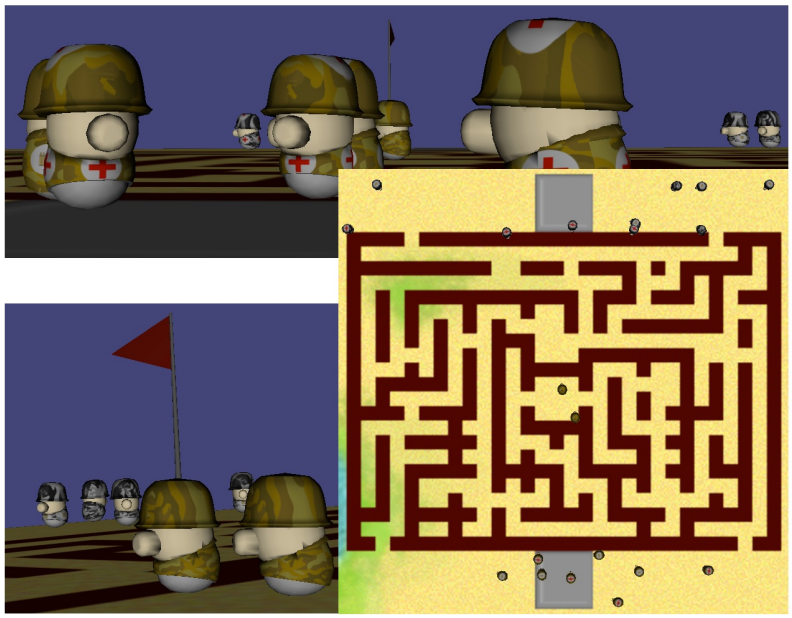
GTI IA Grupo de Tecnología Informática
Inteligencia Artificial

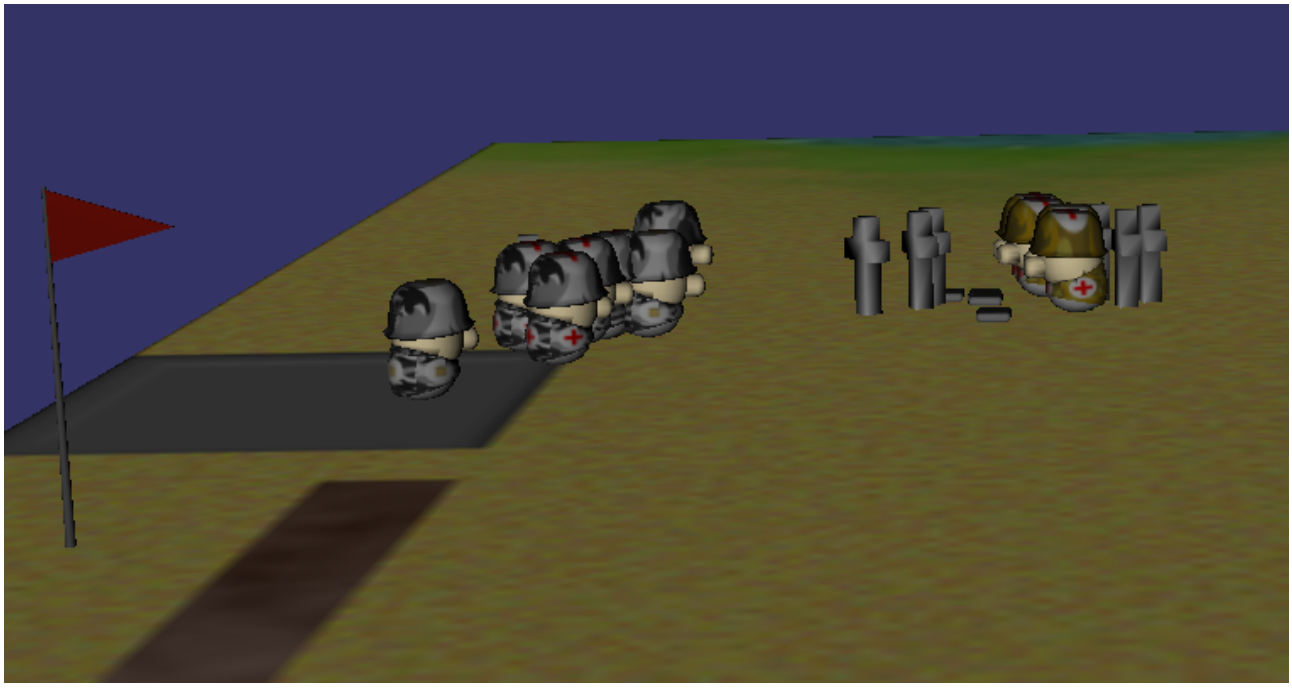
JASON - JGOMAS

Manual de JASON-JGOMAS



1. Introducción

Este documento presenta un Sistema Multi-Agente (SMA) simulador social basado en JADE y Jason que ha sido desarrollado para cumplir diferentes propósitos. El principal es servir como punto inicial para un estudio de la viabilidad de la integración de SMA y Realidad Virtual (RV). De esta manera, combina un visor gráfico 3D con un SMA que puede ser usado en una evaluación cualitativa de la ejecución actual del SMA.
Así, JGOMAS (Game Oriented Multi-Agent System, based on JADE and Jason) es una plataforma de prueba para estudiar la integración completa entre SMA y RV, permitiendo también ser usada como una herramienta de validación de SMA.
Como simulación del entorno en el SMA desarrollado se ha elegido un juego del tipo capturar la bandera. En esta clase de juegos, dos equipos (rojo y azul, aliados y eje) deben competir para capturar la bandera del oponente. Esta modalidad de juego ha llegado a ser un estándar en casi todos los juegos multi-jugador que han aparecido desde Quake.
Es muy fácil e intuitivo aplicar SMA a este tipo de juegos, porque cada soldado puede ser visto como un agente. Más aún, los agentes de un equipo deben cooperar entre ellos para conseguir el objetivo del equipo, compitiendo con el otro equipo.
En nuestra versión de capturar la bandera, los dos equipos participantes son los aliados y el eje. Los agentes aliados deben ir a la base del eje, capturar la bandera y volver a su base, en cuyo caso ganarán el juego. Por otro lado, los agentes del eje deben defender su bandera y, si es capturada, deben regresarla a la base. La partida tiene una duración máxima, transcurrida la cual, si la bandera no ha llegado a la base aliada, el equipo del eje ganará el juego.



2. Arquitectura de Jason - JGOMAS

JGOMAS está compuesto principalmente de dos subsistemas. Por una parte, hay un Sistema Multi-Agente con dos clases diferentes de agentes ejecutándose. Uno de estos tipos de agentes controla la lógica actual del juego, mientras que los otros pertenecen a uno de los dos equipos, y estarán jugando el juego completo. Realmente, este subsistema es una capa que funciona por encima de una plataforma de sistema multi-agente, específicamente JADE, así puede tomar ventaja de todos los servicios que JADE provee.
Jason-JGOMAS, que es la versión actual de la plataforma JGOMAS, permite el uso de agentes Jason para formar los equipos de agentes participantes.
Por otro lado, se ha desarrollado un visor gráfico (*Render Engine*) ad-hoc para visualizar un entorno virtual 3D. De acuerdo a los requerimientos propios de aplicaciones gráficas (alto coste computacional por cortos períodos), este visor gráfico ha sido diseñado como un módulo externo (y no como un agente). Ha sido escrito en C++ usando la biblioteca gráfica OpenGL.
La Figura 1 muestra una vista de la arquitectura de JGOMAS, donde todos sus componentes y sus relaciones pueden ser vistos: la plataforma JADE como el soporte del Sistema Multi-Agente JGOMAS, la cual está compuesta de agentes, uno de ellos actuando como controlador por el resto de agentes, y como interfaz para la aplicación de visor gráfico.

El Sistema Multi-Agente de JGOMAS puede ser visto como un *kernel* (paquete básico), que provee una interfaz para el visor gráfico para establecer una conexión con el juego actual.

2.1. Taxonomía de Agentes

Podemos establecer la siguiente taxonomía de agentes dentro de JGOMAS de acuerdo a su funcionalidad:

- Agentes Internos: son los que son propios de la gestión de la plataforma JGOMAS. Sus comportamientos están predefinidos, y el usuario no puede cambiarlos. Estos agentes son agentes JADE, y existen los siguientes tipos:
  - Manager: Éste es un agente especial. Su objetivo principal es coordinar el juego actual. Además, debe contestar a peticiones del resto de agentes. Otra tarea de la que se encarga es de proveer de una interfaz para los visores gráficos. Por lo tanto, cualquier instancia del visor gráfico puede conectarse al juego actual y mostrar el entorno virtual 3D.
  - Pack: Existen tres tipos distintos de *packs*, los *medic packs* (usados para dar salud a los agentes), *ammo packs* (usados para dar munición a los agentes) y los *objective packs*, esto es, la bandera a capturar. Todos estos agentes son creados y destruidos dinámicamente a excepción de la bandera (sólo existe una bandera durante todo el juego y que no puede ser destruida).
- Agentes Externos: Son los jugadores del juego actual. Tienen un conjunto de comportamientos predefinidos básicos que el usuario puede modificar o incluso añadir nuevos. Estos agentes se deben desarrollar en Jason. Un agente puede jugar un único rol durante el juego actual. Hay tres roles definidos, aunque el usuario puede definir nuevos, cada uno proveyendo un servicio único. Así, estos agentes, o *tropo agents*, se especializan en los siguientes tres roles:
  - Soldier: provee un servicio de *backup* (el agente va a ayudar a compañeros de su equipo).
  - Medic: provee un servicio de *medic* (el agente va a dar *medic packs*).
  - FieldOps: provee un servicio de *ammo* (el agente va a dar *ammo packs*).

Un dominio como el de capturar la bandera permite de una forma sencilla y entretenida establecer un campo de pruebas para algoritmos y optimizaciones a nivel individual en cada agente, así como a nivel de estrategias cooperativas y competitivas entre los distintos equipos.
Los agentes externos se encuentran integrados en el entorno virtual, eso permite la interacción entre ellos (por medio de la percepción de compañeros / enemigos cercanos) lo que puede llevar a la cooperación / coordinación con compañeros del mismo equipo. Además, al estar situados en este entorno virtual, deberán tener en cuenta las características del terreno en el que se encuentren (dificultad de movimiento, paredes…).
Toda la comunicación que se realiza entre los agentes que componen la plataforma se realiza por medio del paso de mensajes según los protocolos establecidos por FIPA ACL.

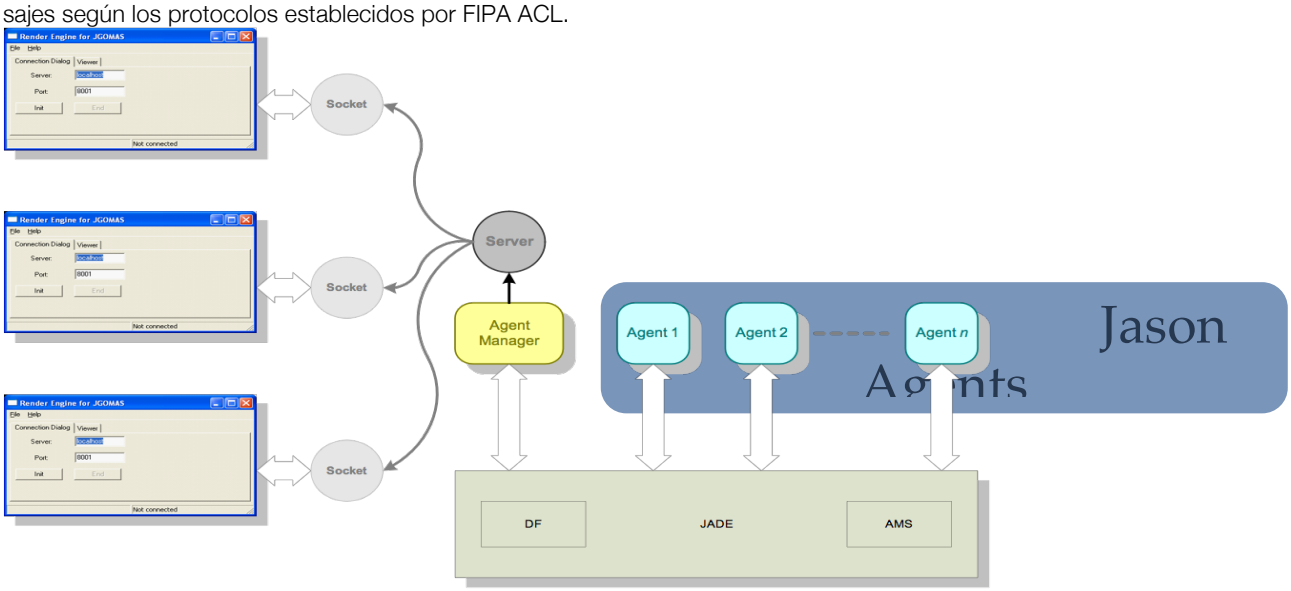


Figura 1: Arquitectura de Jason-JGOMAS

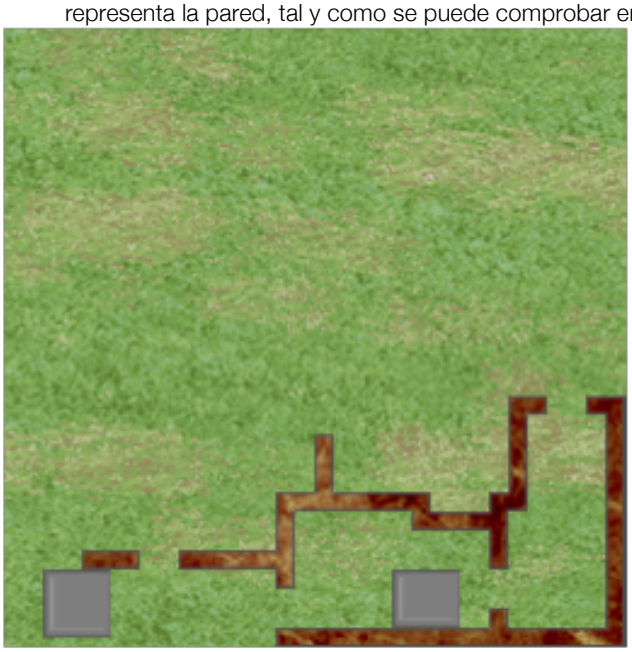
2.2. Mapas

JGOMAS puede usar diferentes mapas para definir su entorno virtual. Estos mapas, por defecto, son de tamaño 256 x 256, con lo que la posición de los agentes vendrá dada por su coordenada (x, y, z), donde x, z tomarán valores entre 0 y 255, mientras que, en los mapas suministrados, la y siempre valdrá 0 (al no tener dichos mapas altura).
Cada agente tiene acceso parcial al mapa donde se desarrolla la partida, puesto que pese a tener acceso a la información estática del mismo, tan sólo puede percibir los objetos que están a cierta distancia (dentro de su "cono de visión").
Los mapas se almacenan en la carpeta **bin\data\maps** de la distribución. En esta carpeta existe una subcarpeta por cada mapa con el nombre **map_XX**, donde **XX** es el número del mapa. En esta carpeta existen diferentes ficheros que definen el mapa. Por ejemplo, el contenido de la carpeta **map_04** es el siguiente:

- map_04_cost.txt: Este fichero define las paredes del mapa por medio del uso de ***** para indicarlo. El contenido de este fichero es el siguiente:

```
********************************
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                         **   **
*                         *    *
*                *        *    *
*                *        *    *
*                *        *    *
*              ********   **    *
*              *      *****     *
*              *          *     *
*    ***   ******         *     *
*              *                *
*                               *
*                               *
*                         *     *
********************************
```

- map_04_terrain.bmp: Este fichero define el aspecto artístico del mapa, tal y como se puede ver en la imagen de la izquierda de la Figura 2.
- map_04_cost.bmp: Este fichero define las paredes del mapa usando una imagen en blanco y negro, donde el blanco representa la pared, tal y como se puede comprobar en la imagen de la derecha de la Figura 2.



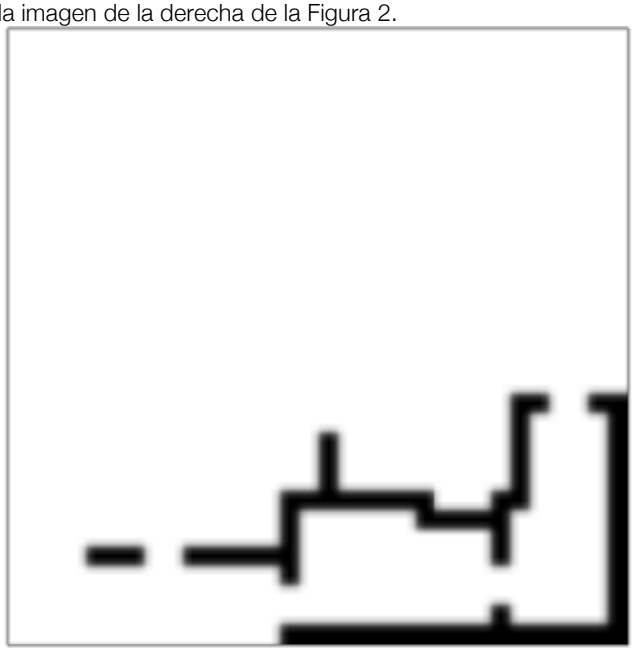


Figura2: Imagen Izquierda: map_04_terrain.bmp - Imagen derecha: map_04_cost.bmp

- map_04.txt: Este fichero contiene la definición de diferentes parámetros de configuración para le SMA y el visor gráfico:

- JADE_OBJECTIVE: Localización inicial de la bandera.
- JADE_SPAWN_ALLIED: Localización de la base Aliada.
- JADE_SPAWN_AXIS: Localización de la base del Eje.
- JADE_COST_MAP: Tamaño y nombre del fichero de costes.
- RENDER_ART_MAP: Tamaño y nombre del fichero de arte.
- RENDER_COST_MAP: Tamaño y nombre del fichero de arte de los costes.
- RENDER_HEIGHT_MAP: Tamaño y nombre del fichero de arte de las alturas.

El contenido de este fichero es el siguiente:

```
[JADE]
JADE_OBJECTIVE: 28 28
JADE_SPAWN_ALLIED: 2 28 4 30
JADE_SPAWN_AXIS: 20 28 22 30
JADE_COST_MAP: 32 32 map_04_cost.txt
[JADE]

[RENDER]
RENDER_ART_MAP: 256 256 map_04_terrain.bmp
RENDER_COST_MAP: 32 32 map_04_cost.bmp
RENDER_HEIGHT_MAP: 32 32 map_04_heightmap.bmp
[RENDER]
```

3. Tareas

Una tarea es algo que un agente tiene que realizar en una posición concreta del entorno virtual. Existen diversos tipos de tareas, de acuerdo a cuales son las diferentes acciones que un agente puede realizar en el entorno virtual, siendo los principales:

- `TASK_GIVE_MEDICPAKS`: Un médico debe generar paquetes de medicina en un lugar determinado (la posición del agente que se lo ha solicitado y al que ha accedido ir a dárselos).
- `TASK_GIVE_AMMOPAKS`: Un fieldops debe generar paquetes de munición en un lugar determinado (la posición del agente que se lo ha solicitado y al que ha accedido ir a dárselos).
- `TASK_GIVE_BACKUP`: Un soldado debe ir a ayudar a un compañero a un lugar determinado (la posición del agente que se lo ha solicitado y al que ha accedido ir a dárselo).
- `TASK_GET_OBJECTIVE`: El agente, del equipo atacante o `ALLIED`, tiene que ir a la posición inicial de la bandera a por ella. Si logra coger la bandera, esta tarea se transforma en ir hasta su base de origen.
- `TASK_GOTO_POSITION`: El agente debe ir a una posición concreta.

Una tarea tiene asociada además de su tipo, el agente que provoca la tarea (el propio agente, o el agente que solicitó que se le diese algo como vida…), la posición donde se debe llevar a cabo, la prioridad y algún posible contenido adicional. Siempre se lanzará la tarea de prioridad más alta. Es posible redefinir la prioridad de cada tipo de tarea.

Las tareas las pone en ejecución el sistema, no el usuario, pudiendo éste tan sólo añadir tareas a la lista de tareas activas del agente:

- JASON: para añadir una tarea se utiliza un plan: **`add_task`**

```
!add_task(task(TaskPriority, TaskType, Agent, Position, Content))
!add_task(task(TaskType, Agent, Position, Content))
```

  Ejemplos (dos opciones):

```
!add_task(task(1000, "TASK_GET_OBJECTIVE", M, pos(ObjectiveX, ObjectiveY, ObjectiveZ),
"",));
!add_task(task("TASK_GET_OBJECTIVE", M, pos(ObjectiveX, ObjectiveY, ObjectiveZ), “”));
```

  Dichos objetivos disparan el plan que crea la tarea, el segundo le asigna la prioridad definida por el agente.

4. Bucle de ejecución

- Cada agente externo de JGOMAS ejecuta una máquina de estados como la que aparece en la Figura 3
  - `STANDING`: El agente no tiene ninguna tarea lanzada.
  - `GO_TO_TARGET`: El agente ha lanzado una tarea y está moviéndose hacia la posición en la que debe realizarla.
  - `TARGET_REACHED`: El agente ha alcanzado la posición en la que debe realizar la tarea lanzada y está realizando las acciones indicadas en la tarea.

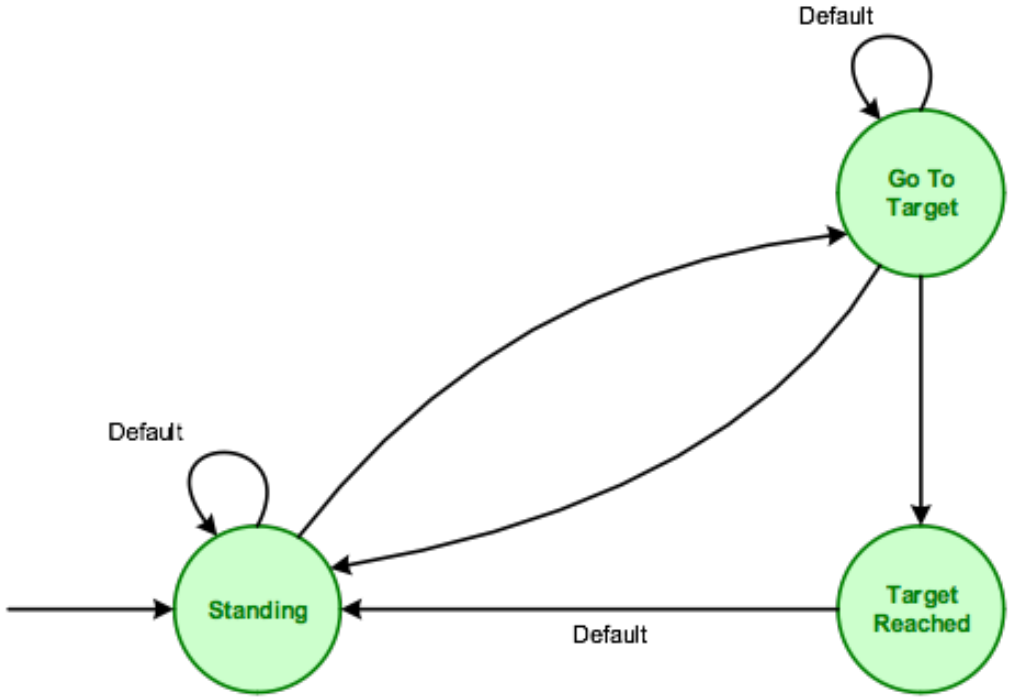


Figura 3: Máquina de estados que define el comportamiento de los agentes externos de JGOMAS

5. Interfaz (API)

La interfaz (API) para el trabajo en jGomas se compone de los ficheros .asl escritos en Jason que contienen los distintos agentes (cada uno con creencias, metas y comportamientos). Dentro de estos ficheros, jgomas.asl incluye los comportamientos no modificables del agente. Para modificar el comportamiento de los agentes se deben modificar los archivos con nombres de la forma jasonAgent_TEAM_TYPE.asl, donde TEAM se refiere al bando al que pertenece el agente (ALLIED, AXIS), y TYPE al tipo del agente (soldado, MEDIC, FIELDOPS). Por tanto, tenemos 6 ficheros que cubren los posibles tipos de agentes:

- `jasonAgent_ALLIED.asl`
- `jasonAgent_ALLIED_MEDIC.asl`
- `jasonAgent_ALLIED_FIELDOPS.asl`
- `jasonAgent_AXIS.asl`
- `jasonAgent_AXIS_MEDIC.asl`
- `jasonAgent_AXIS_FIELDOPS.asl`

En cuanto a creencias de los agentes, las principales son:

- **`tasks(task_list)`:** Contiene la lista de tareas activas del agente. Ejemplo:

```
tasks([task(1000,"TASK_GET_OBJECTIVE","Manager",pos(224,0,224),""),task(1001,
"TASK_WALKING_PATH","A2",pos(204,0,228),"")])
```

En este ejemplo la lista tiene dos tareas activas:

  - `task(1000,"TASK_GET_OBJECTIVE","Manager",pos(224,0,224),"")` que indica que debe ir a por el objetivo (la bandera) que está en la posición `pos(224,0,224).` La prioridad asignada a esta tarea es 1000.
  - `task(2000,"TASK_GIVE_MEDIPAKS","A2",pos(204,0,228),"")` que indica que debe ir a la posición `pos(204,0,228)` donde está el agente A2 esperando paquetes de medicina (se supone que el agente que tiene esta tarea activa es médico). La prioridad asignada a esta tarea es 2000.

- **`fovObjects(object_list)`:** Contiene la lista de objetos que ahora mismo ve el agente. La estructura de un objeto es `[#, TEAM, TYPE, ANGLE, DISTANCE, HEALTH, POSITION].`

Ejemplo: `[1,200,1,0.58,14.76,78,pos(214,0,219)],` el objeto 1 es del equipo 200 (AXIS), del tipo 1 (agente),

con el ángulo 0,58, a una distancia de 14,76, que tiene una salud de 78 y su posición es `pos(214,0,219).`
Nota: Los valores de TEAM son: 100 (Allied), 200 (Axis), 1003 (bandera).

- **`state(Estado_actual):`** Esta creencia se utiliza para indicar el estado del agente en su máquina de estados: standing, eligiendo que tarea hacer o esperando; `go_to_target`, acudiendo a su próximo objetivo; `target_reached`, ha llegado al objetivo; `quit`, debe terminar.
- **`my_health(X):`** Guarda la salud del agente. El valor inicial y máximo es 100, cuando llega a 0, el agente se muere.
- **`my_ammo(X):`** Guarda la cantidad de balas que dispone el agente. El valor inicial es 100.
- **`my_position(X,Y,Z):`** Guarda la posición última conocida por el agente.

Los diferentes planes sobrecargables con los que cuentan los agentes son los siguientes:

- **`!perform_look_action`**
  Este objetivo se invoca cuando se ha mirado alrededor y se supone que se ha actualizado la lista de objetos alrededor `fovObjects(L)`. Sería necesario implementar el plan asociado a la creación de este evento para poder ver qué hay alrededor.
- **`!perform_aim_action`**
  Si hay un enemigo al que apuntar se lanza este objetivo, el cual puede servir para tomar alguna decisión respecto a qué hacer, ir a por él, pasar de él, etc. Una implementación sencilla del plan asociado ya está disponible. Sería interesante mejorar dicho plan asociado para tomar una decisión de a quien apuntar más refinada.
- **`!get_agent_to_aim`**
  Este objetivo se invoca después del `!perform_look_action`, se utilizaría para decidir si hay algún enemigo al que apuntar. Una implementación sencilla del plan asociado ya está disponible. Sería interesante mejorar dicho plan asociado para tomar una decisión de a quien apuntar más refinada.
- **`!perform_no_ammo_action`**
  Este objetivo se lanza cuando el agente dispara y no le quedan balas. Sería necesario implementar el plan asociado a la creación de este evento para tomar una decisión, por ejemplo huir.
- **`!perform_injury_action`**
  Este objetivo se lanza cuando el agente es disparado. Sería necesario implementar el plan asociado a la creación de este evento para tomar una decisión, por ejemplo huir si hay poca vida.
- **`!performThresholdAction`**
  Este objetivo se lanza cuando el agente dispone de menos vida o balas que los umbrales definidos en `my_ammo_threshold(X)` y en `my_health_threshold(X)`. Una implementación sencilla del plan asociado ya está disponible, la cual siempre pide ayuda a médicos o a fieldops de su equipo. Sería interesante mejorar dicho plan asociado para tomar una decisión más refinada de qué hacer.
- **`!setup_priorities`**
  Este objetivo se lanza en la inicialización del agente para fijar las prioridades de las tareas del agente. Cada agente puede tener sus propias prioridades. Una implementación sencilla del plan asociado ya está disponible. Sería interesante en ocasiones modificarlo para añadir nuevas tareas o cambiar las prioridades para tener agentes que se comportan de forma distinta.
- **`!update_targets`**
  Este objetivo puede utilizarse para actualizar las tareas y sus prioridades. Se invoca cuando el agente pasa a estado standing y debe elegir nueva tarea entre las disponibles. Sería necesario implementar el plan asociado a la creación de este evento.

6. Comunicación

Es necesario resaltar que un agente asume un único rol durante toda la partida, si un agente A1 es creado como soldado de tipo Soldier y pertenece al bando de los AXIS, lo será durante toda la partida.

Cada rol tiene unas características y ofrece unos determinados servicios básicos que pueden ser mejorados. La indicación del rol y de los servicios básicos que ofrece un agente se realiza en la inicialización, mediante un proceso que se conoce como Registro. No obstante un agente puede añadir nuevos servicios durante el desarrollo del juego.

6.1. Registro

Un rol debe registrar un servicio para que el resto de roles puedan solicitarlo. El registro se realiza mediante la función interna: **`.register( "JGOMAS", "type")`**

En este ejemplo se registraría el servicio "`type`" por parte del agente que invocase la función. El registro se utiliza para registrar en el DF un servicio del tipo "type" por parte del agente que lo ejecuta.

Ej: `.register( "JGOMAS", "medic_AXIS")`; registra en el DF el servicio "`medic_AXIS`".

Existe un conjunto de servicios ya existentes según el rol que juega el agente soldado:

- En el caso de los agentes de tipo ALLIED todos los agentes registran de forma transparente que pertenecen a dicho bando para que el resto de jugadores de su equipo lo puedan saber.

```
.register( "TEAM", "ALLIED");
```

En el caso de agentes aliados del tipo soldado, se registra el servicio "`backup_ALLIED`":

```
.register( "JGOMAS", "backup_ALLIED");
```

En el caso de agentes aliados del tipo médico, se registra el servicio "`medic_ALLIED`":

```
.register( "JGOMAS", "medic_ALLIED");
```

En el caso de agentes aliados del tipo fieldop, se registra el servicio "`fieldops_ALLIED`":

```
.register( "JGOMAS", "fieldops_ALLIED");
```

- En el caso de los agentes de tipo AXIS todos los agentes registran de forma transparente que pertenecen a dicho bando para que el resto de jugadores de su equipo lo puedan saber.

```
.register( "TEAM", "AXIS");
```

En el caso de agentes aliados del tipo soldado, se registra el servicio "`backup_AXIS`":

```
.register( "JGOMAS", "backup_AXIS");
```

En el caso de agentes aliados del tipo soldado, se registra el servicio "`medic_AXIS`":

```
.register( "JGOMAS", "medic_AXIS");
```

En el caso de agentes aliados del tipo soldado, se registra el servicio "`fieldops_AXIS`":

```
.register( "JGOMAS", "fieldops_AXIS");
```

Una vez un agente ha registrado un servicio, es posible que otro agente del mismo equipo consulte que agentes hay de su equipo que ofrecen un determinado servicio.

¿Cómo saber qué servicios hay disponibles desde un agente?

Para ello utilizaremos la acción interna **`.my_team(type,list)`**

Esta acción devuelve la lista de agentes del equipo al que pertenece el agente que la invoca disponibles a través de las páginas amarillas (DF) que son del tipo "`type`" (se excluye de la lista a él mismo).

En el ejemplo: `.my_team("medic_AXIS", E)`
Suponiendo que es ejecutado por un agente del equipo "AXIS" devolvería la lista E que contiene los médicos vivos del equipo.

De esta forma es posible por ejemplo, saber el nombre de los agentes de un equipo. Para ello se debe invocar la función interna:

```
.my_team("AXIS", E);   ó
.my_team("ALLIED", E);
```

A partir de ahí, recorriendo la lista E, el agente puede realizar las acciones que estime oportunas.

Ejemplo de uso

```
…
.my_team("AXIS", E1);
.my_name(Me);
.println("Mi equipo es: ", E1, " y yo soy: ", Me );
.length(E1, X);
 if (X==0) {  .println("Me he quedado sólo");  }
```

En este caso un agente consulta qué agentes hay en su equipo y si la lista está vacía es que se ha quedado sólo.

6.2. Coordinación

JGOMAS dispone de mecanismos que permiten la coordinación entre agentes. Ésta puede ser de dos tipos:

- Sin comunicación (implícita): Se consigue mediante la sensorización del entorno por parte de una agente. Cuando un agente mira a su alrededor se lanza el objetivo `!perform_look_action`. Reescribiendo el plan asociado se puede decidir qué hacer en función de lo que se ve.
- Con comunicación (explicita): En este caso se utiliza el paso de mensajes mediante la acción interna `.send_msg_with_conversation_id`

```
.send_msg_with_conversation_id (Rec, Perf, Cont, ConvId)
```

Donde:

- Rec → receptor del mensaje (puede ser una lista)
- Perf → performativa (tell, untell, achieve…)
- Cont → contenido
- ConvId → Id de conversación (se usa en Jade)

Ejemplo de uso: A1 quiere enviar un mensaje a su equipo diciendo que vayan a su posición (para ayudar, para coordinarse, para reagruparse…)

El código sería el siguiente:

```
…
?my_position(X,Y,Z);
.my_team("AXIS", E1);
.concat("goto(",X, ", ", Y, ", ", Z, ")", Content1);
.send_msg_with_conversation_id(E1, tell, Content1, "INT");
```

NOTA: “INT” es un identificador de conversación inventado, se recomienda su uso ya que existen ciertos identificadores internos que si fuesen usados podrían provocar un funcionamiento no deseado del agente.

Siguiendo el mismo ejemplo, el resto de agentes del equipo dispondrían de un plan de la forma:

```
+goto(X,Y,Z)[source(A)]
 <-
    .println("Recibido un mensaje de tipo goto de ", A);
 .
```

en este caso el agente que recibe el mensaje sólo imprime el mensaje recibido por pantalla, pero se podría hacer algo más sofisticado, como cambiar la tarea a realizar por el agente que recibe el mensaje. En este caso el código sería:

```
+goto(X,Y,Z)[source(A)]
```

```
<-
   .println("Recibido mensaje goto de ", A);
   !add_task(task("TASK_GOTO_POSITION", A, pos(X, Y, Z), ""));
   -+state(standing);
   -goto(_,_,_) .
```

7. Anexo: Instalación plugin para Eclipse de Jason

7.1. Instalación

Para instalar el plug-in de Jason para Eclipse deben seguirse estos pasos y además tener la versión 3.7.0 (Indigo) o superior de Eclipse.

Paso 1

Descarga la última versión de Jason en el siguiente enlace: `http://sourceforge.net/projects/jason/files/`

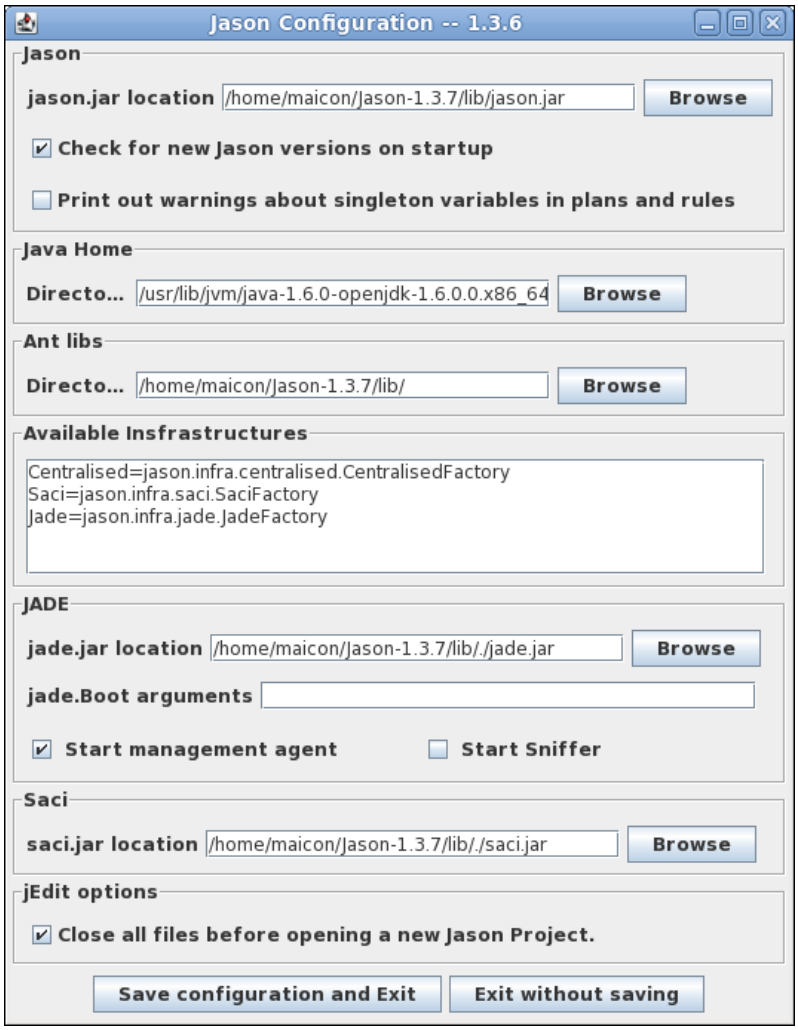
Paso 2

Después de la descarga, descomprímela en cualquier directorio de tu máquina.

Paso 3

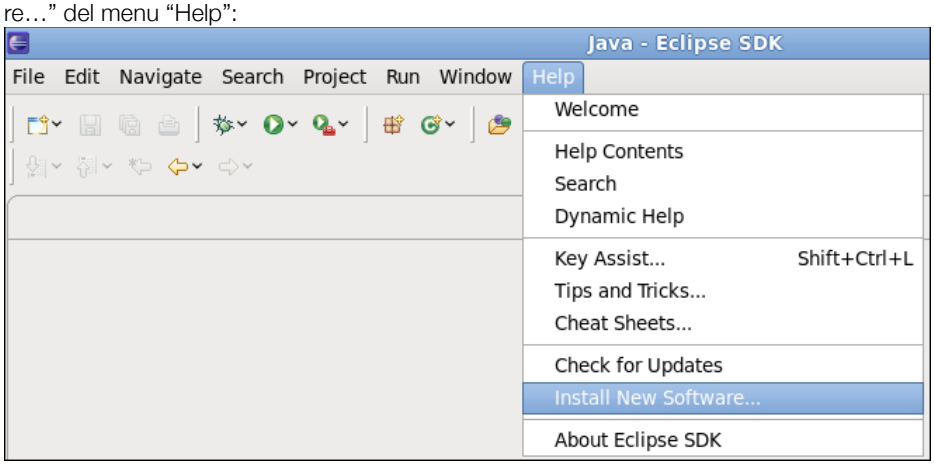
Si nunca has ejecutado Jason en tu ordenador, ejecuta el archivo "lib/jason.jar" haciendo doble clic sobre él. También puedes ejecutar este archivo con el siguiente comando: `java -jar lib/jason.jar`

La siguiente figura muestra la ventaja que debe aparecer al ejecutar el archive jason.jar. Asegúrate de los directorios de las librerías. Sólo deberías tener que cambiar el directorio "Java Home", el resto se rellenan automáticamente pero puedes cambiarlos si quieres.



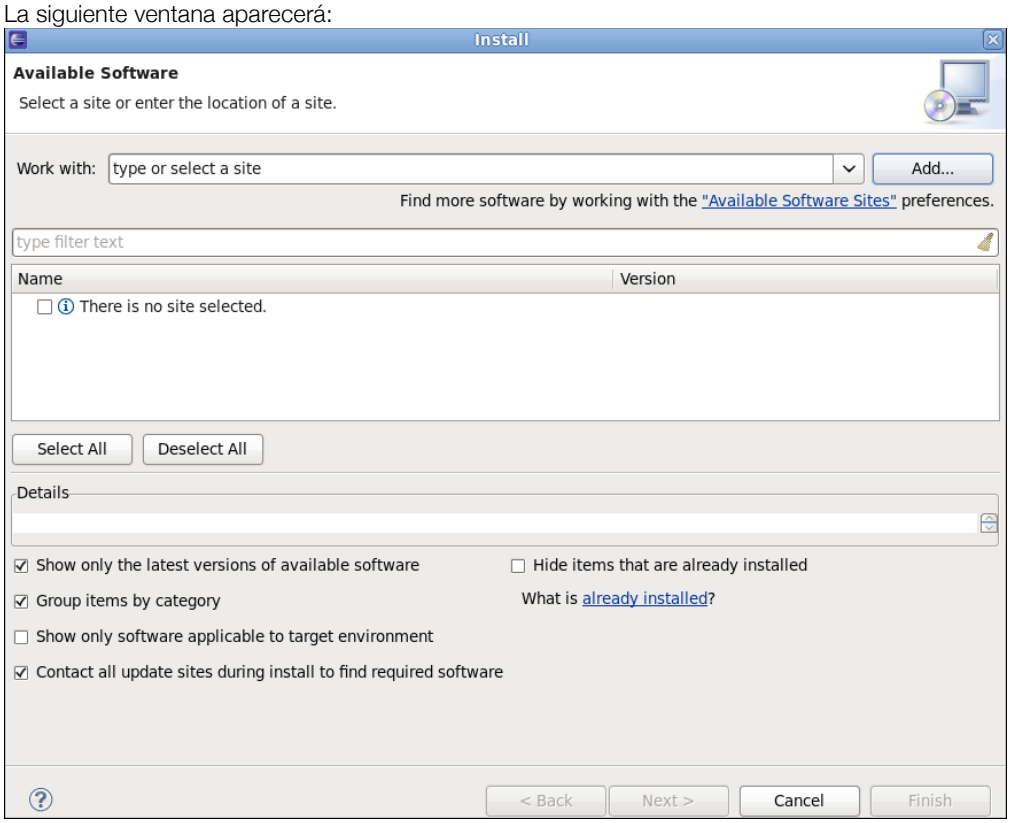
Paso 4

Finalmente, puedes instalar el plugin de Jason para Eclipse abriendo Eclipse y seleccionando la opción “Install New Software…” del menu “Help”:



Paso 5

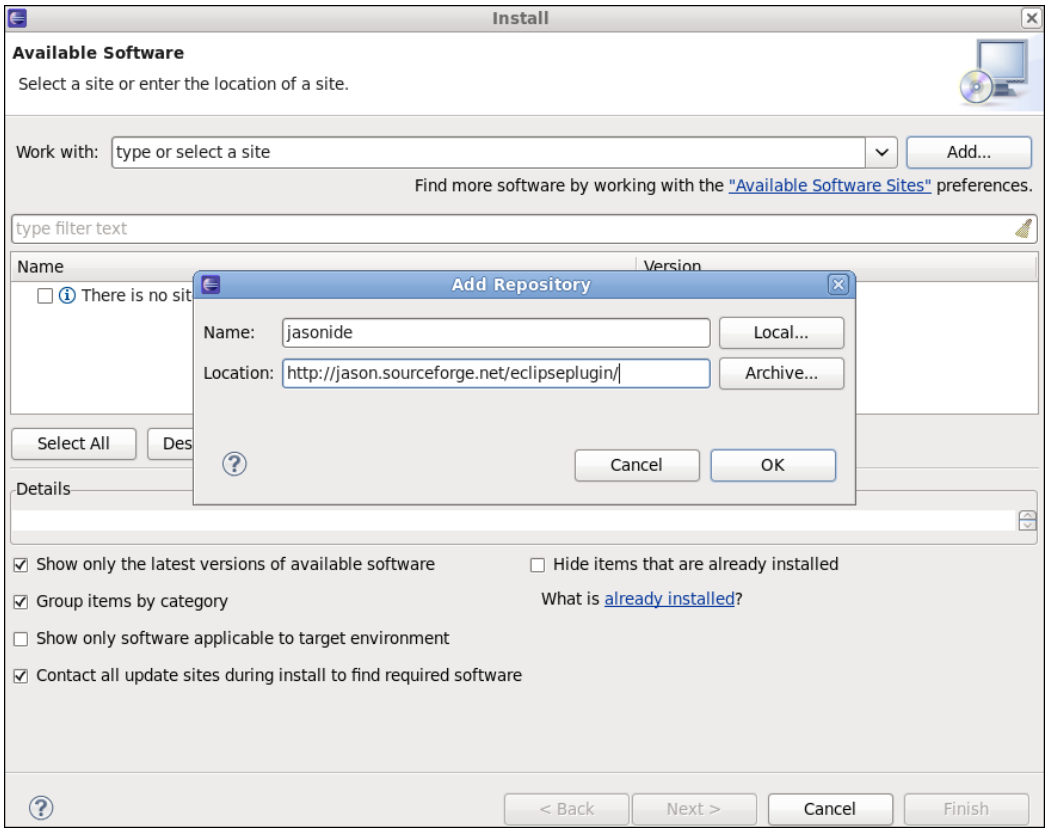
La siguiente ventana aparecerá:



Paso 6

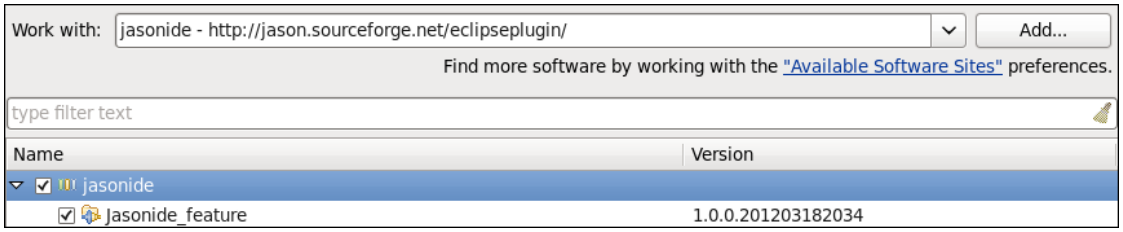
Haz clic en el botón “Add” y rellena el formulario como se muestra en la figura siguiente. Los parámetros son:

Location (Eclipse Indigo): `http://jason.sourceforge.net/eclipseplugin/`
Location (Eclipse Juno/Kepler): `http://jason.sourceforge.net/eclipseplugin/juno/`
Para terminar, haz clic en el botón “OK”.



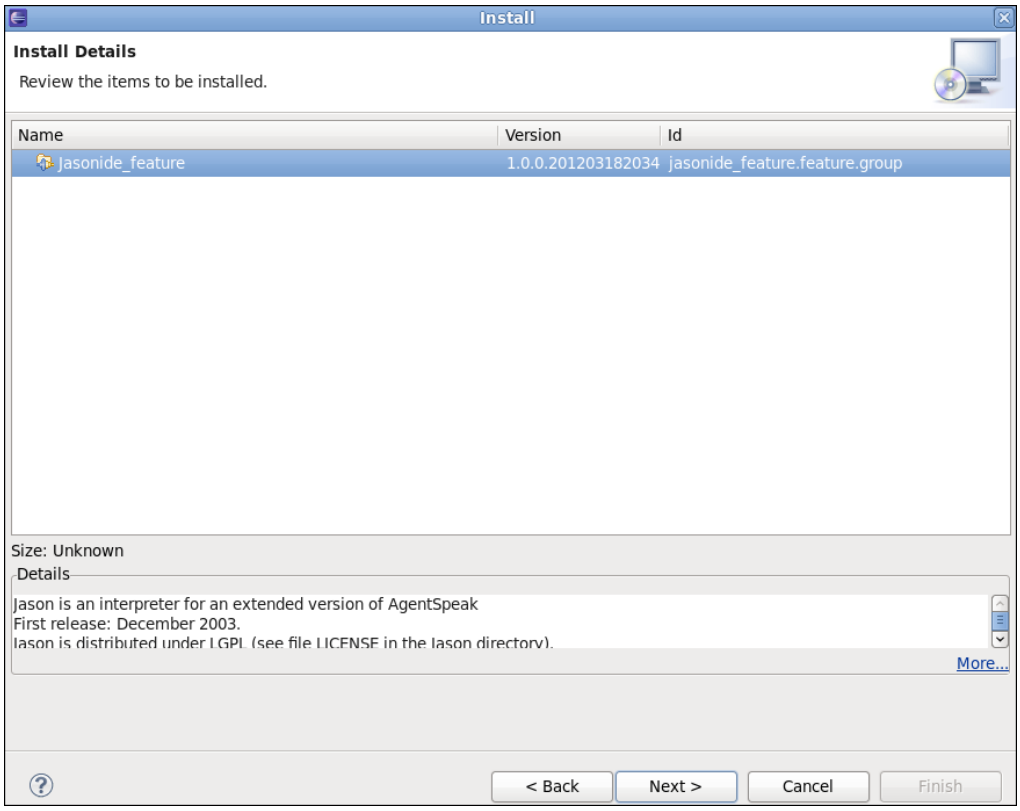
Paso 7

Marca la opción “jasonide” y pulsa el botón “next”. Espera un momento mientras Eclipse busca dependencias.



Paso 8

En la ventana siguiente pulsa de nuevo el botón “next”.

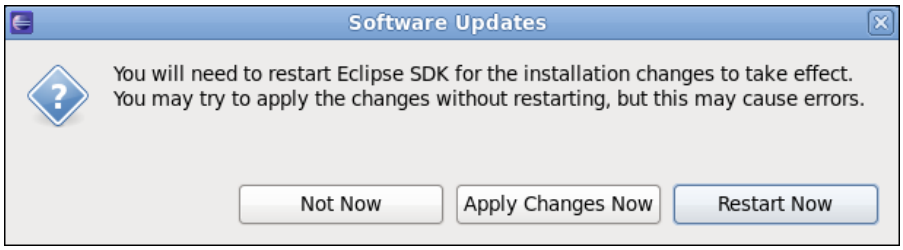


Paso 9

La última ventana que se muestra es la de licencia. Marca la opción "I accept the terms of the license agreements" y pulsa el botón "finish". En ese instante, empezará la instalación que podrá prolongarse durante unos minutos. Por favor, espera.

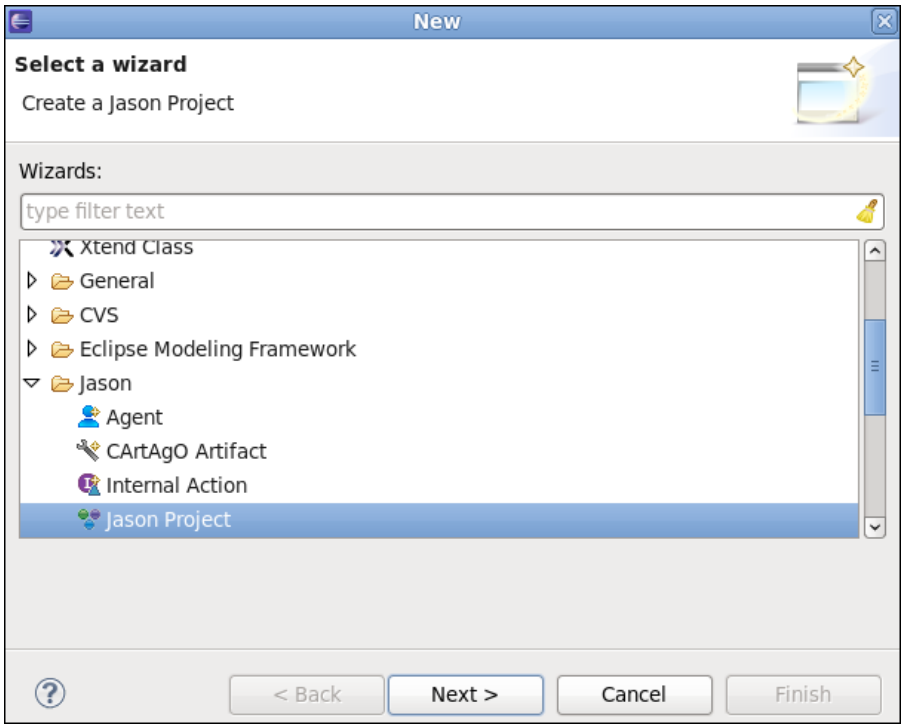
Paso 10

Al final del proceso aparecerá una ventana para terminar la instalación. Elige la opción "Restart Now".



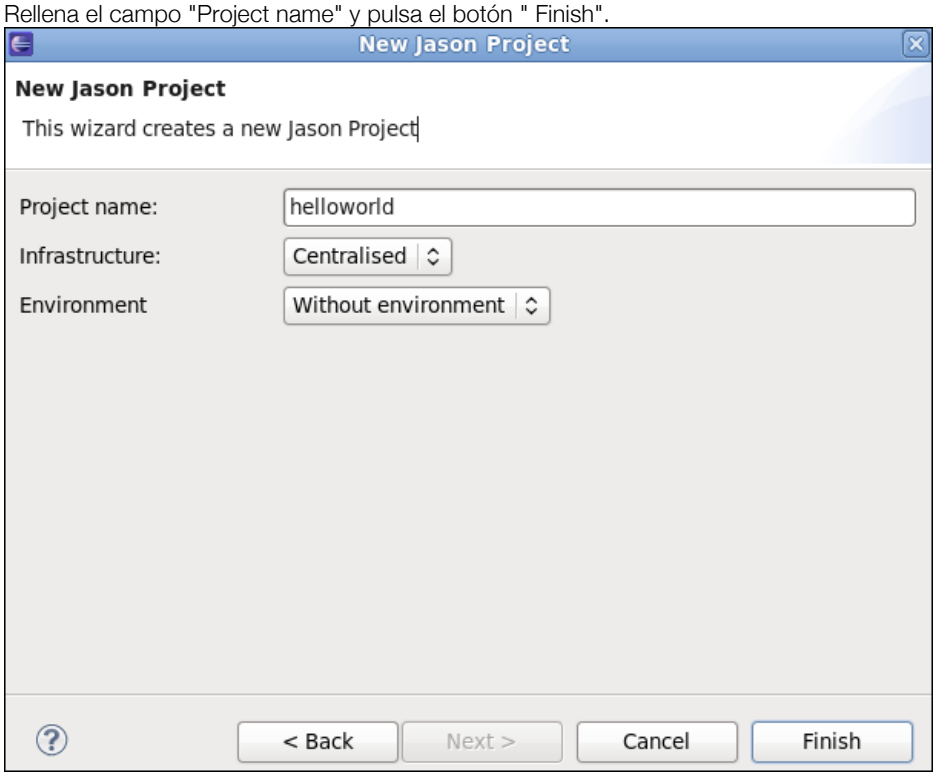
Paso 11

Ahora ya está todo listo. Para comprobar la instalación del plug-in, crea un simple proyecto 'hola mundo'. Puedes hacerlo desde el menú (File > New > Jason Project) o (File > New > Other > Jason > Jason Project).



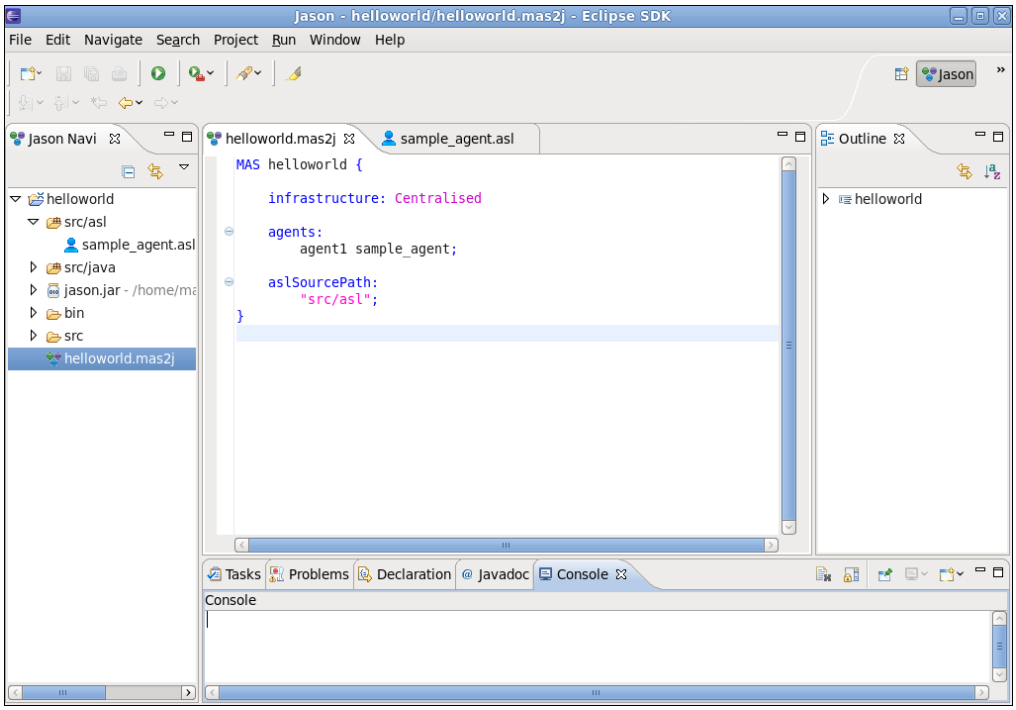
Paso 12

Rellena el campo "Project name" y pulsa el botón " Finish".



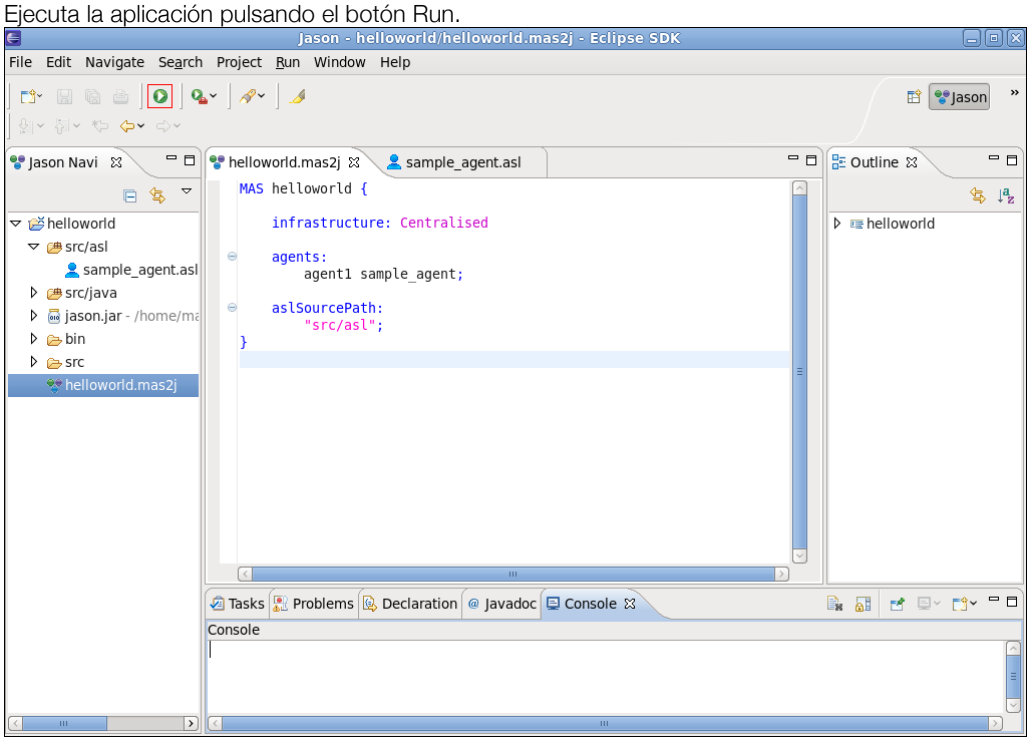
Paso 13

Si todo funciona bien, ¡habrás creado tu primer proyecto!



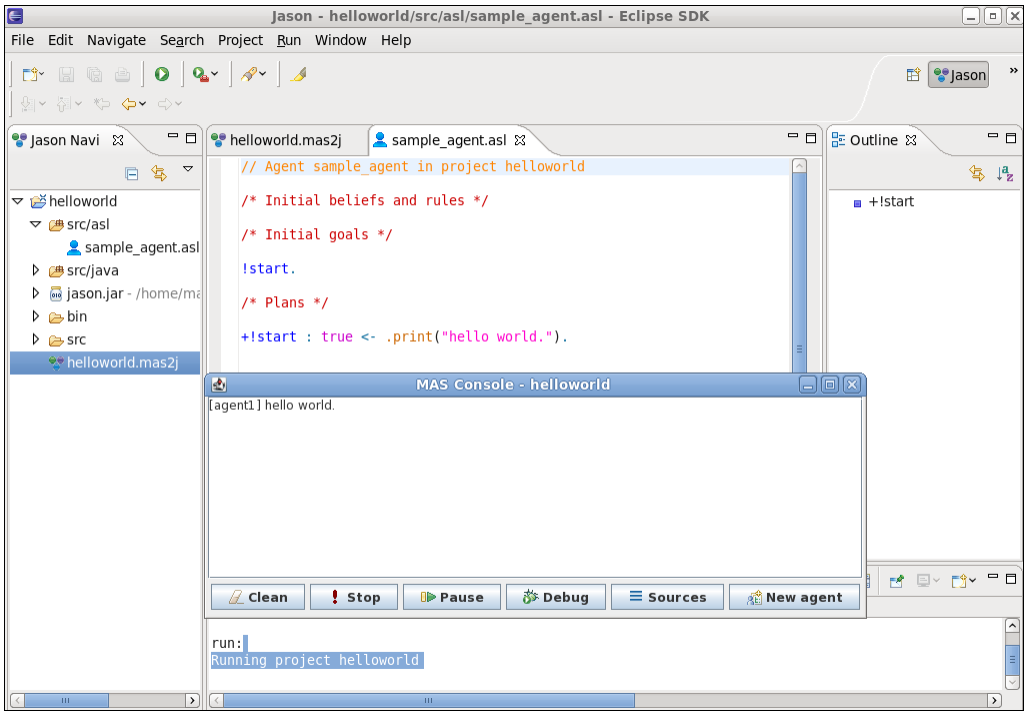
Paso 14

Ejecuta la aplicación pulsando el botón Run.



Paso 15

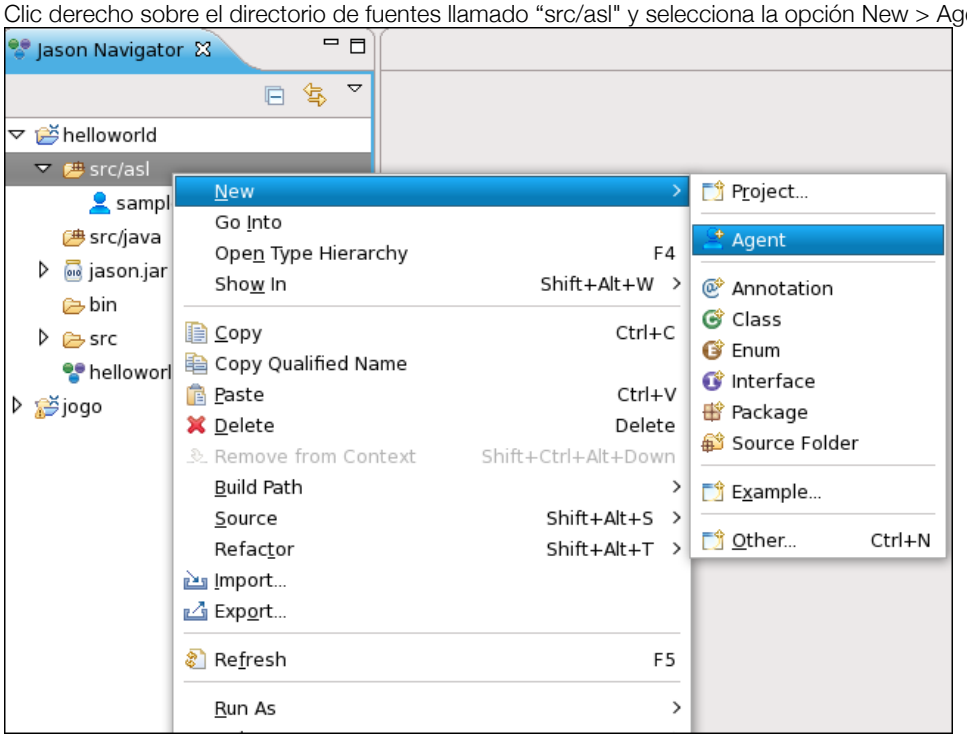
El resultado de la ejecución será un mensaje "hello world" en pantalla.



7.2. Cómo crear un agente

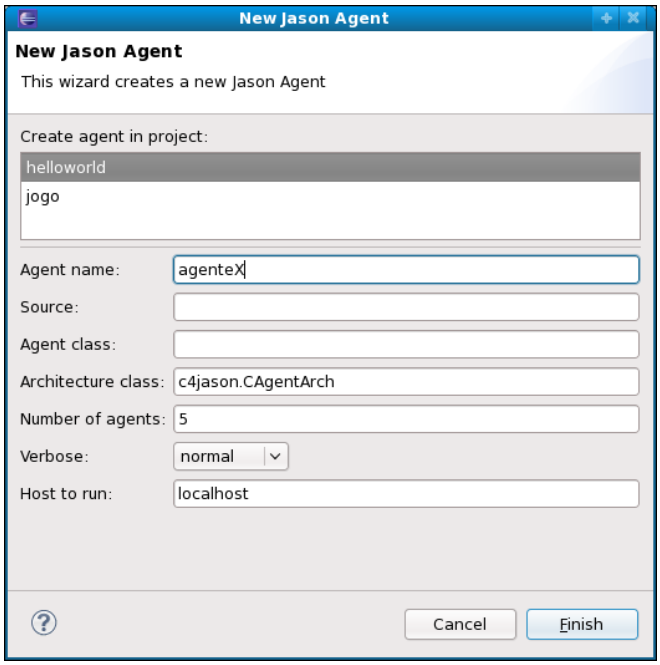
Paso 1

Clic derecho sobre el directorio de fuentes llamado “src/asl" y selecciona la opción New > Agent.



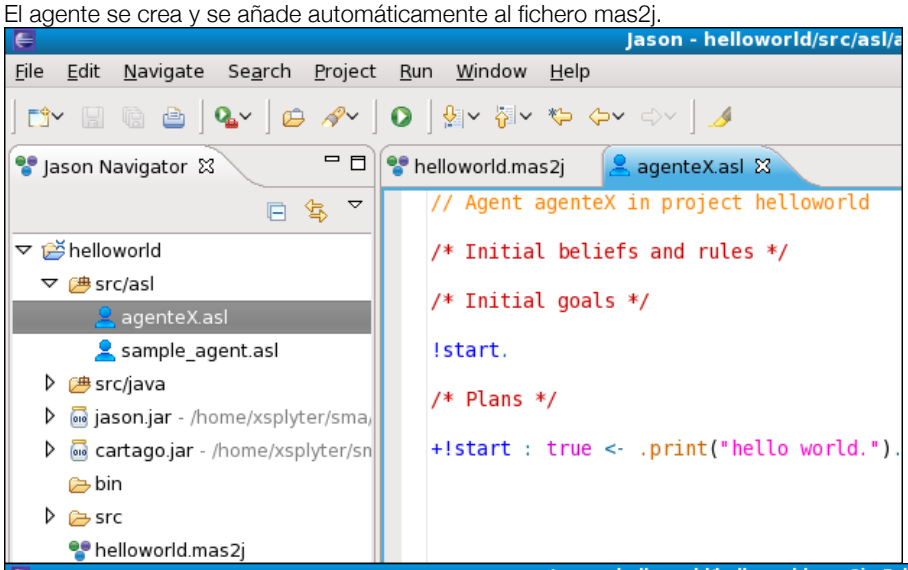
Paso 2

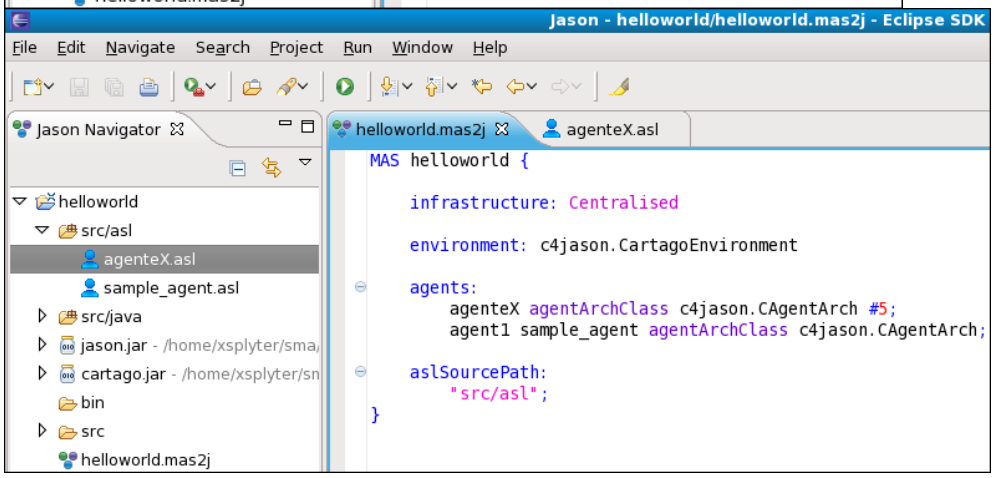
Rellena el formulario. El único campo requerido es el nombre del agente. Después, pulsa el botón “Finish”.



Paso 3

El agente se crea y se añade automáticamente al fichero mas2j.

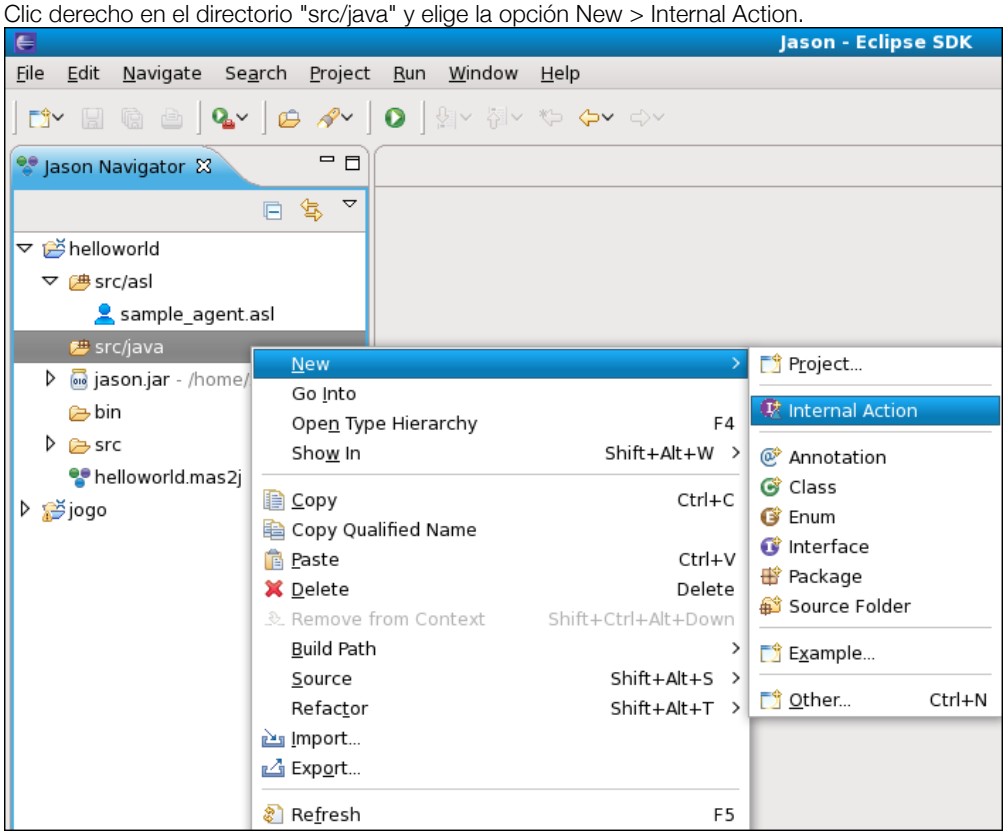




7.3. Como crear una acción interna

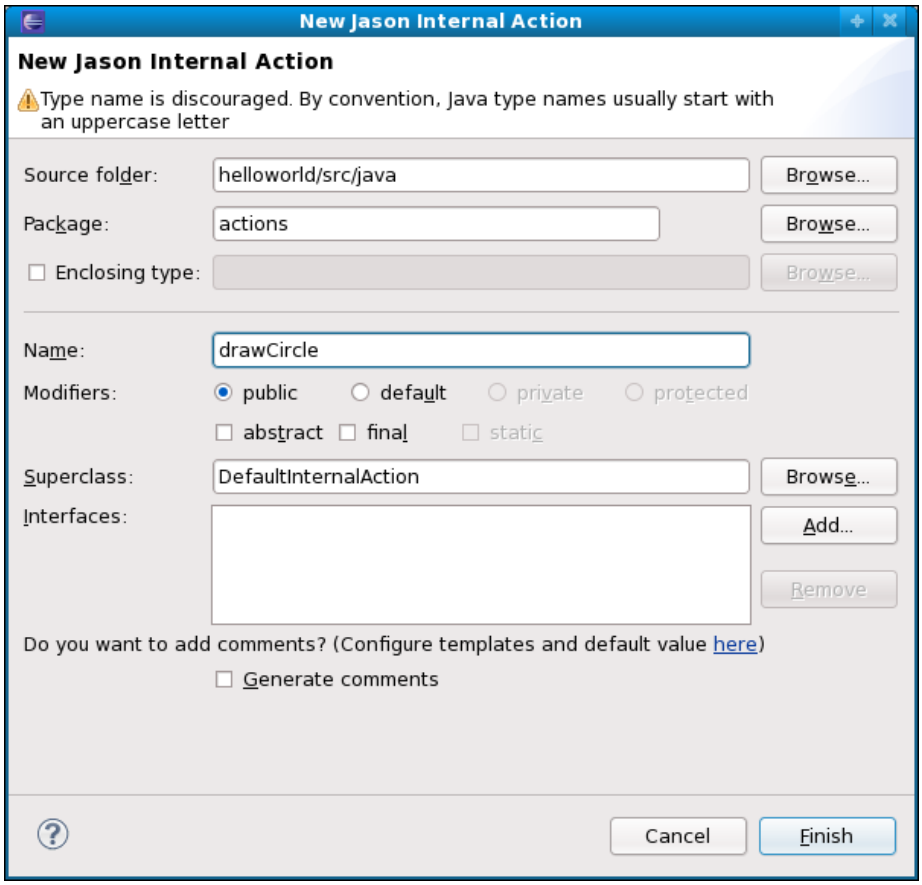
Paso 1

Clic derecho en el directorio "src/java" y elige la opción New > Internal Action.



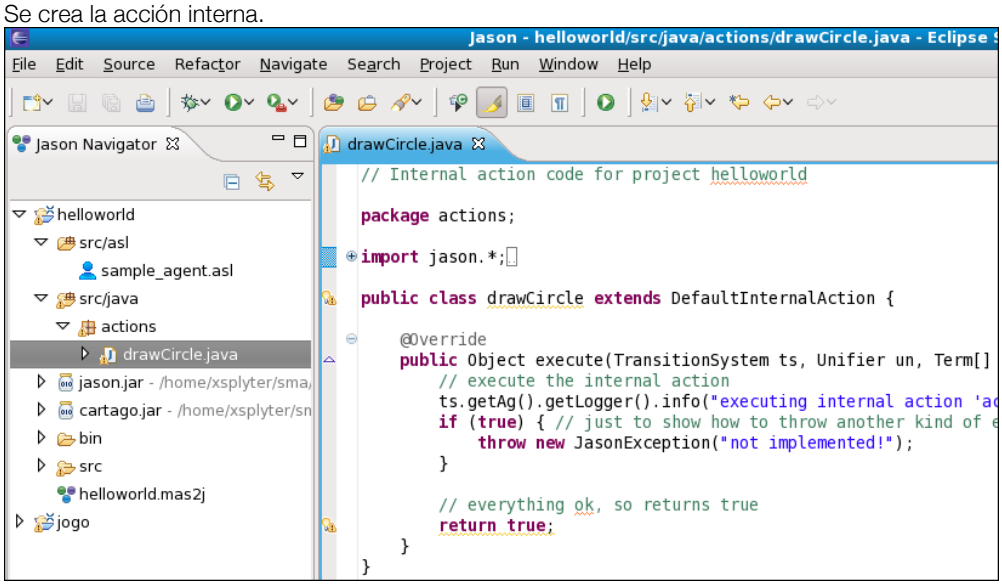
Paso 2

Rellena el formulario. Una acción interna es una clase java, por lo que sólo se require el nombre de la clase.
Nota: se sugiere dar un nombre cuya primera letra esté en minúsculas y también se recomienda dar un nombre al paquete.



Paso 3

Se crea la acción interna.

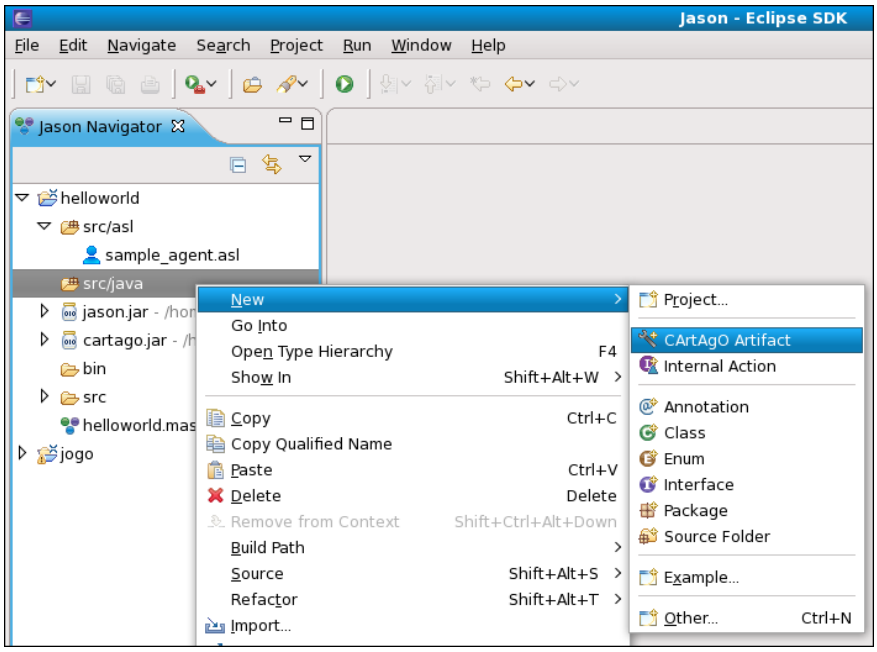


7.4. Cómo crear un artefacto

Nota: Un artefacto CArtAgO solo se utiliza si se selecciona CArtAgO como entorno para el MAS.

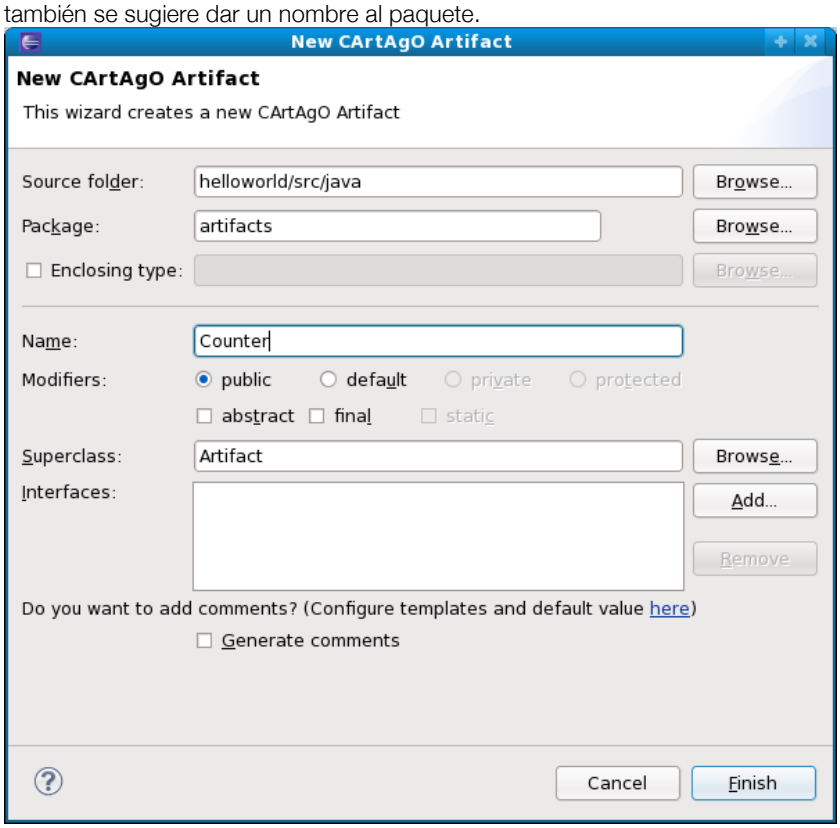
Paso 1

Clic derecho en el directorio "src/java" y selecciona la opción New > CArtAgO Artifact.



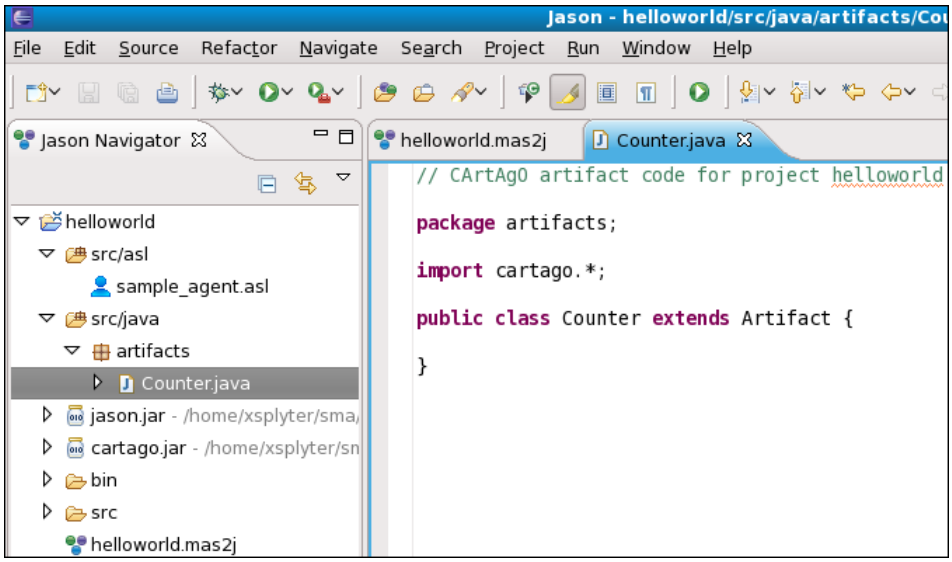
Paso 2

Rellena el formulario. Un artefacto CArtAgO es una clase java, solo se require el nombre de la clase.
Nota: a diferencia de las acciones internas, en este caso se debe usar un nombre con la primera letra en mayúsculas, y también se sugiere dar un nombre al paquete.



Paso 3

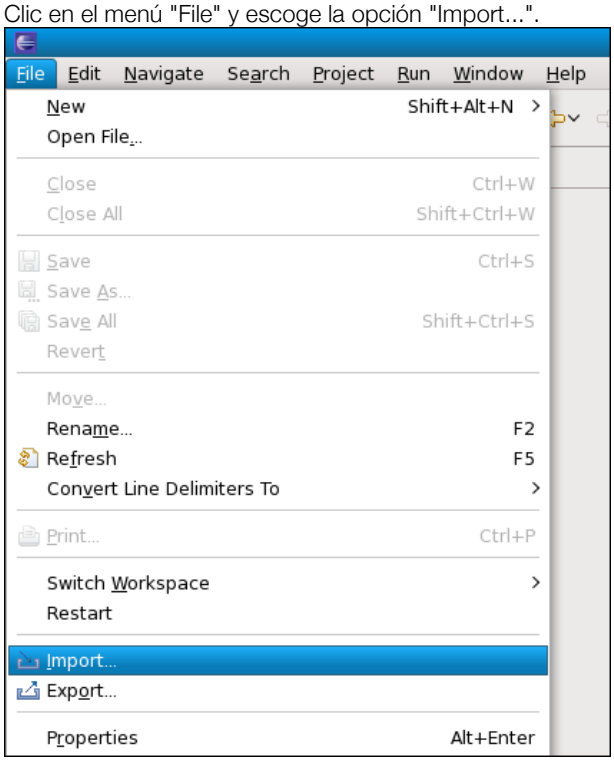
El artefacto CArtAgO se ha creado.



7.5. Cómo importar un proyecto Jason

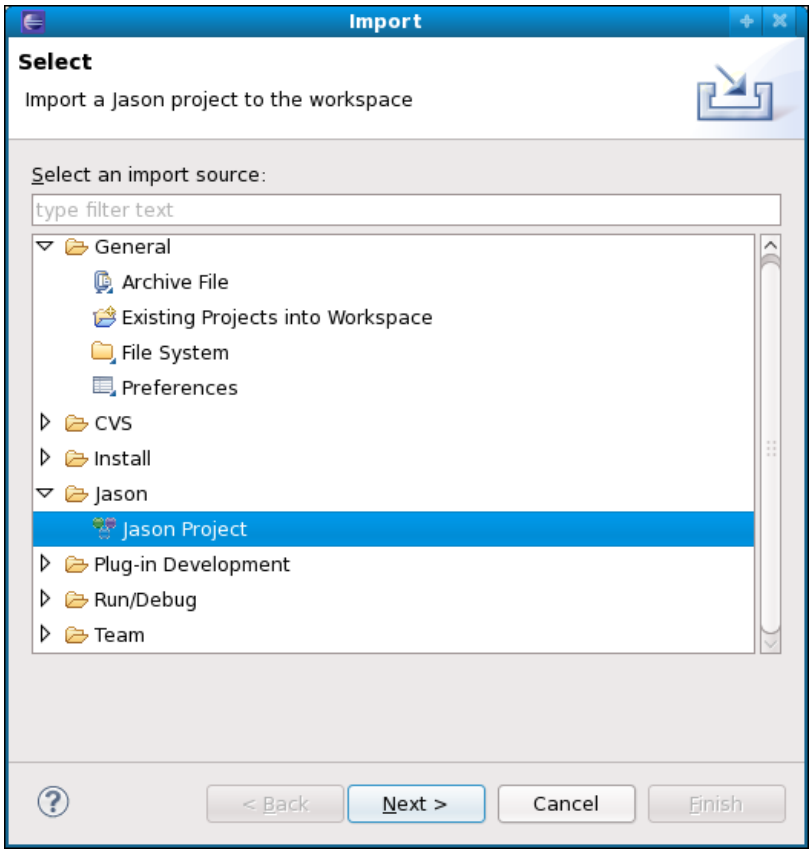
Paso 1

Clic en el menú "File" y escoge la opción "Import...".



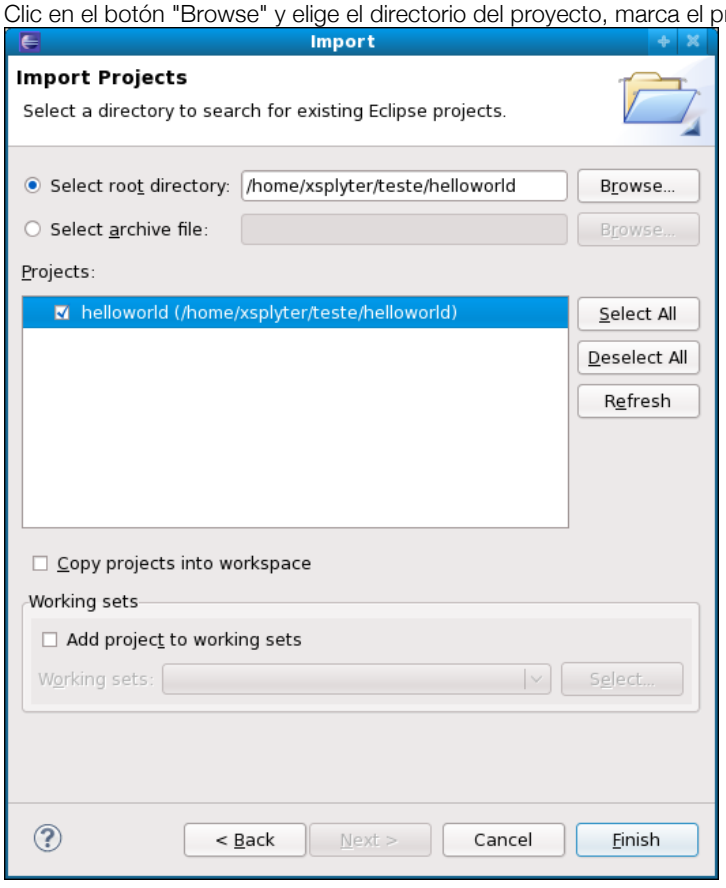
Paso 2

Selecciona la opción Jason > Jason Project.



Paso 3

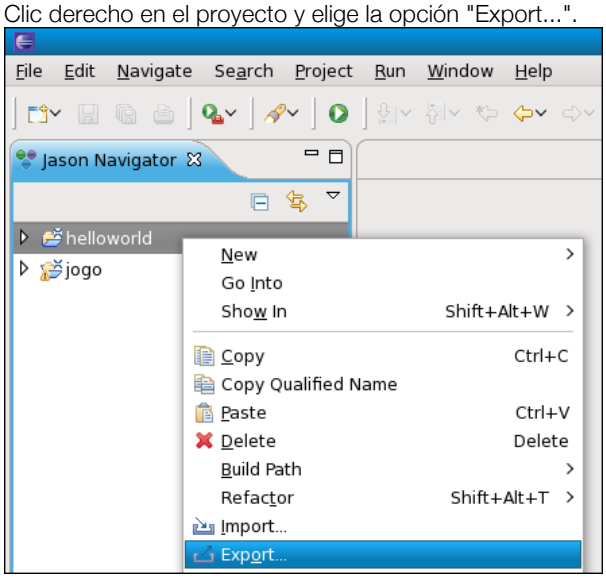
Clic en el botón "Browse" y elige el directorio del proyecto, marca el proyecto que quieras importar y pulsa el botón "Finish".



7.6. Cómo exportar un proyecto Jason

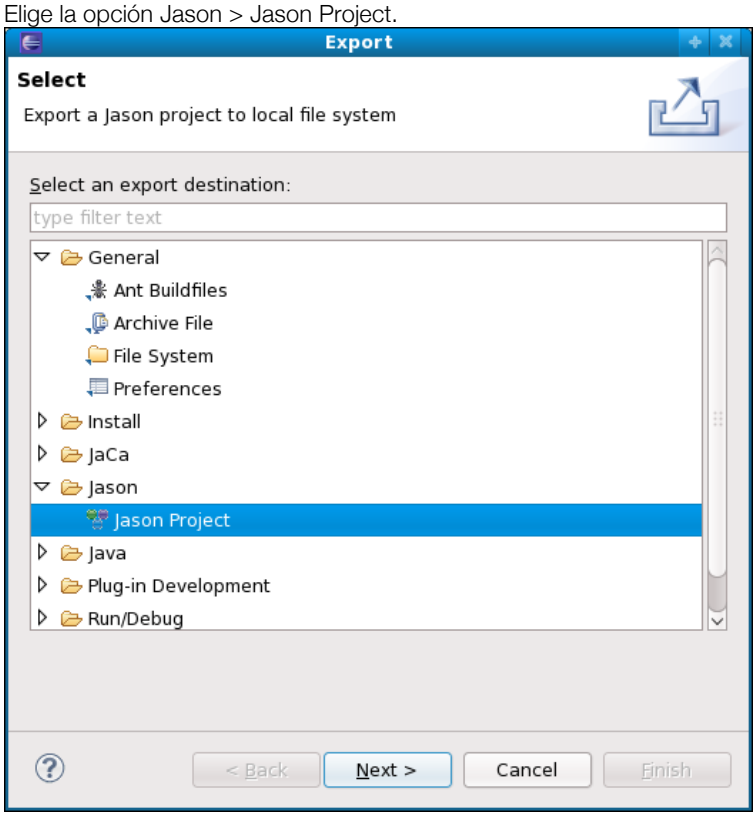
Paso 1

Clic derecho en el proyecto y elige la opción "Export...".



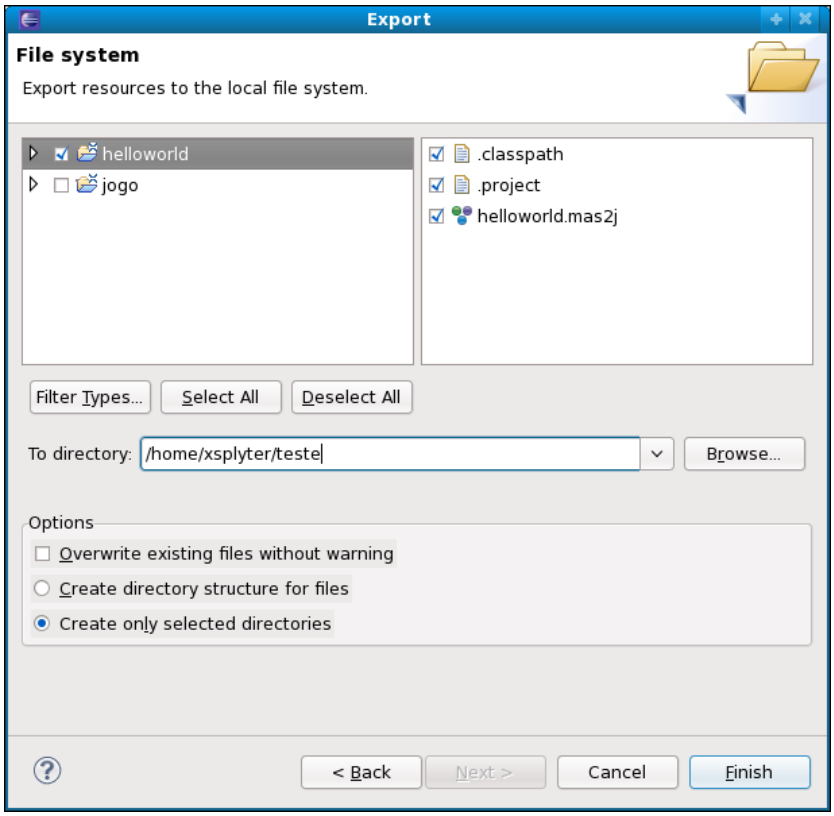
Paso 2

Elige la opción Jason > Jason Project.



Paso 3

Clic en el botón "Browse", elige el directorio donde quieras exportar el proyecto y pulsa "Finish".

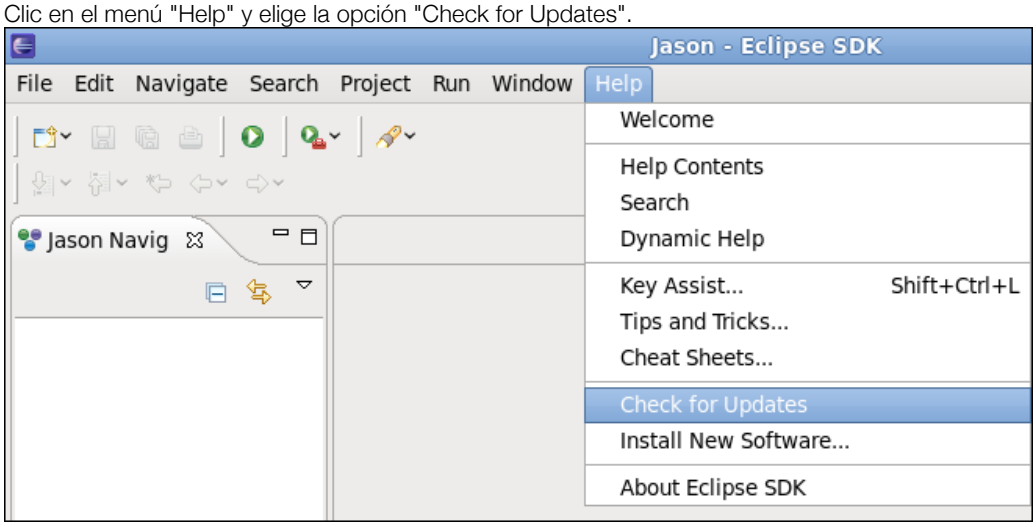


7.7. Cómo actualizar el plugin de Jason

Existen dos formas para actualizar el plugin de Jason para Eclipse.

Primer método

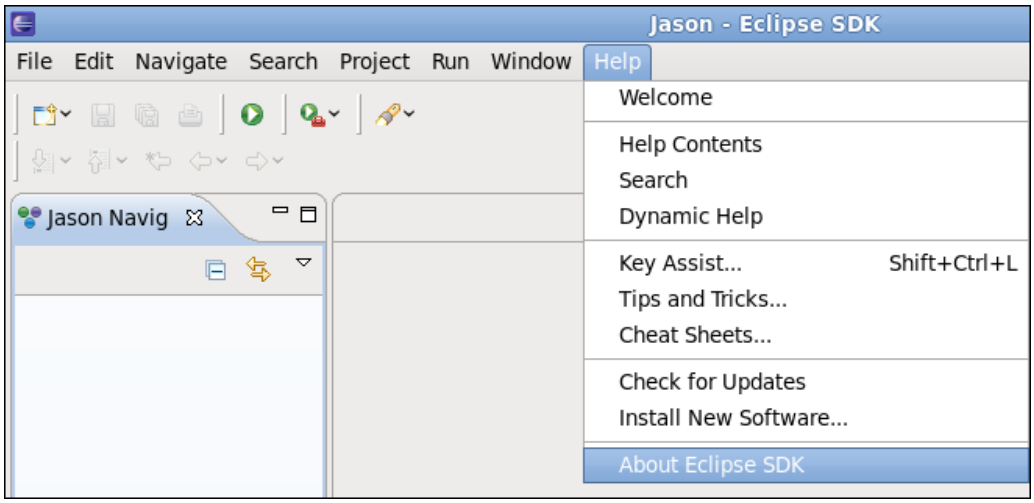
Clic en el menú "Help" y elige la opción "Check for Updates".



Segundo método

Paso 1

Clic en el menú "Help" y elige la opción "About Eclipse SDK".

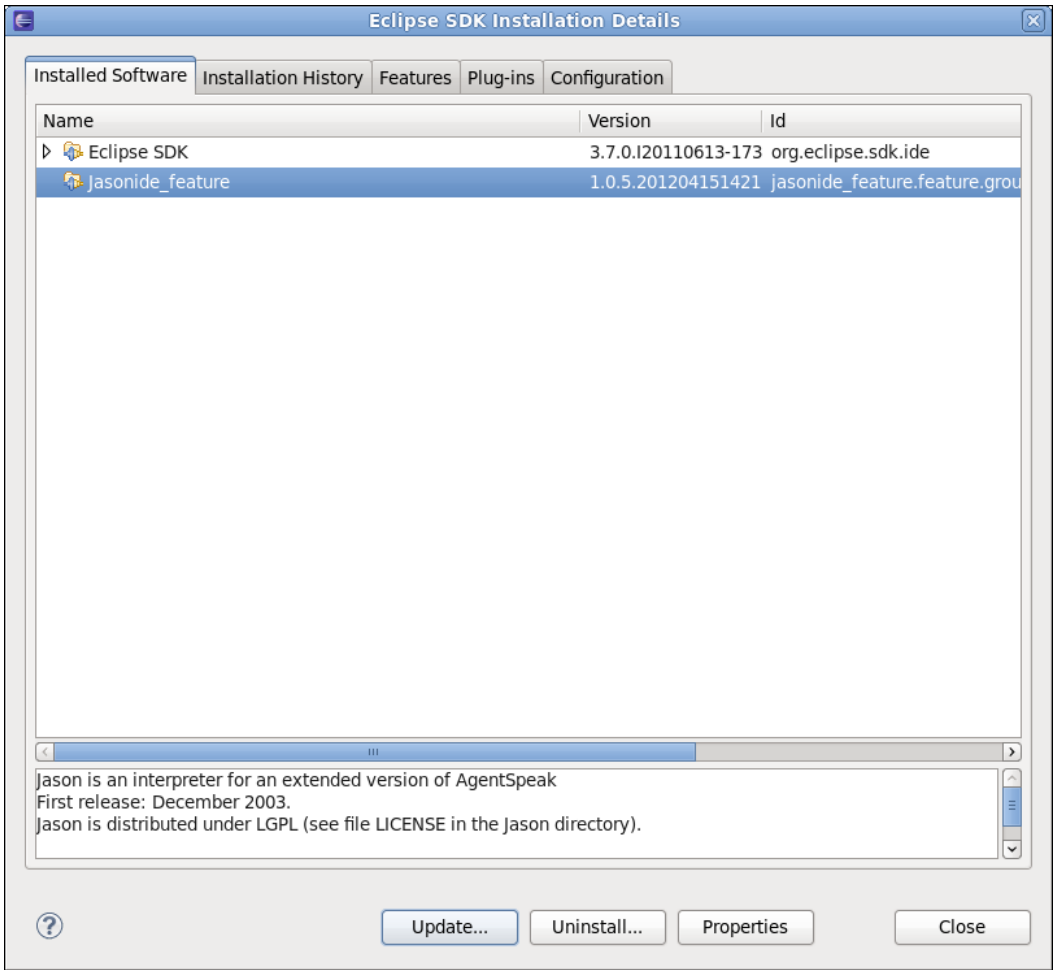


Paso 2

Pulsa el botón "Installation Details".

Paso 3

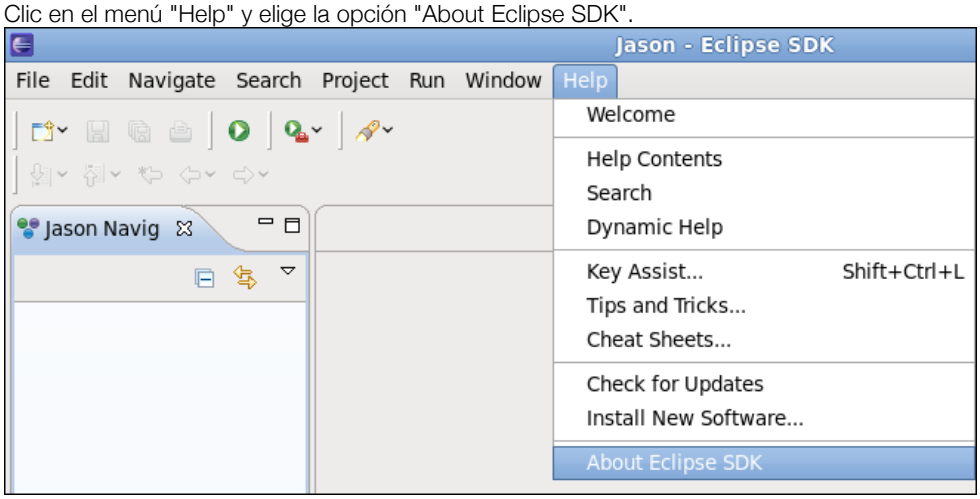
Selecciona "jasonide_feature" y haz clic en el botón "Update...".



7.8. Cómo desinstalar el plugin de Jason para Eclipse

Paso 1

Clic en el menú "Help" y elige la opción "About Eclipse SDK".

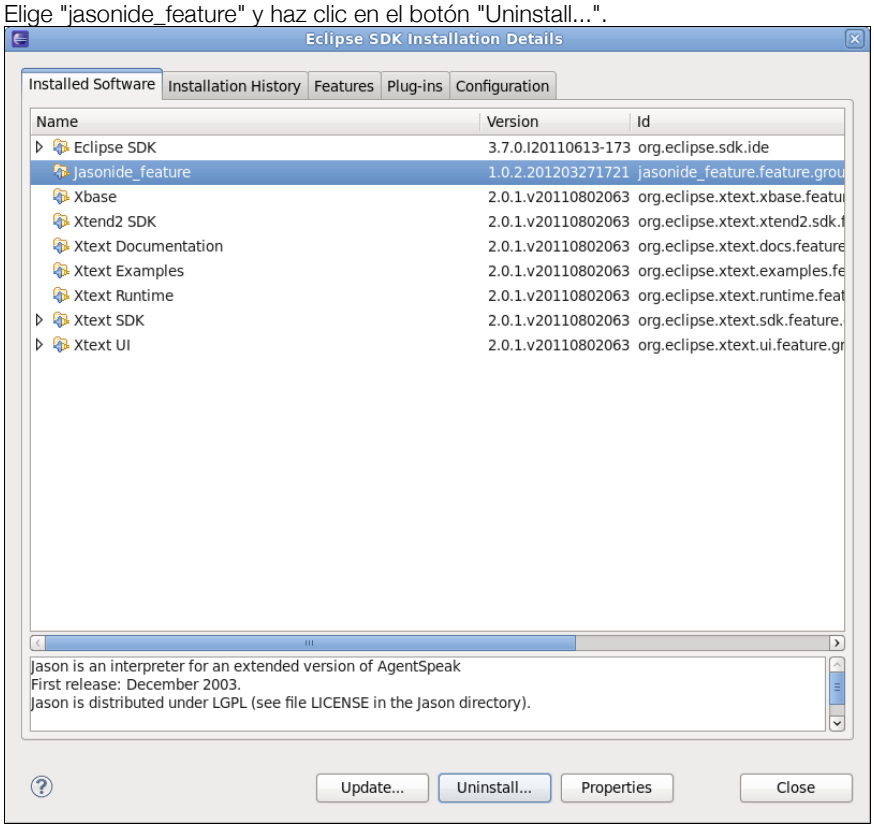


Paso 2

Pulsa el botón "Installation Details".

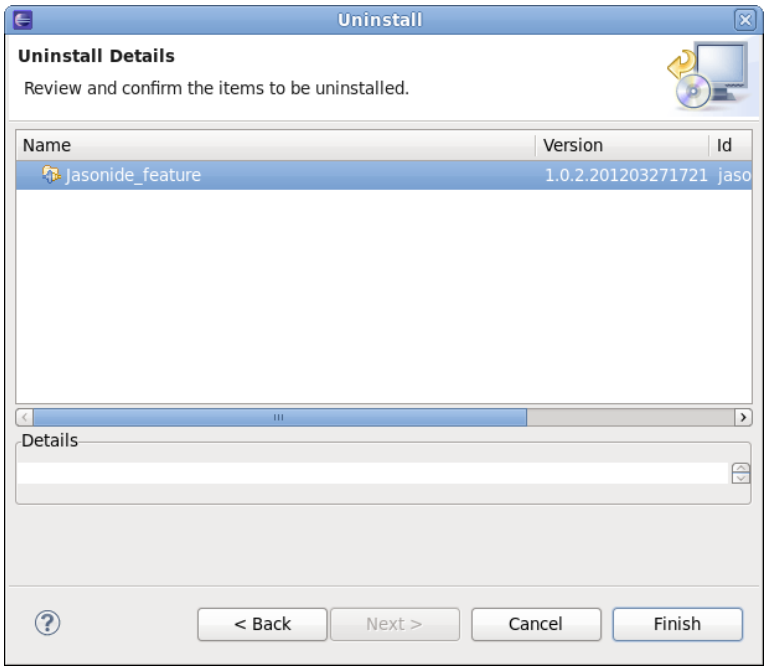
Paso 3

Elige "jasonide_feature" y haz clic en el botón "Uninstall...".



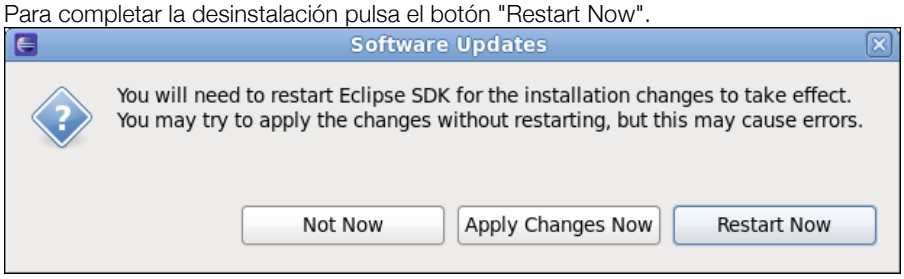
Paso 4

Confirma el proceso de desinstalación pulsando el botón "Finish".



Paso 5

Para completar la desinstalación pulsa el botón "Restart Now".



Más información en `http://jason.sf.net`.

8. Anexo: Ejecución de JGOMAS

Para ejecutar JGOMAS primero es necesario lanzar el manager y a continuación los agentes.

- Lanzar el mánager: Ejecutar el siguiente comando o lanzar el archivo `jgomas_manager.bat` (Windows) o `jgomas_manager.sh` (Linux):

```
java –classpath "lib\jade.jar;lib\jadeTools.jar;lib
\Base64.jar;lib\http.jar;lib\iiop.jar;lib
\beangenerator.jar;lib\jgomas.jar;student.jar;lib
\jason.jar;lib\JasonJGomas.jar;classes;.” jade.Boot -gui
“Manager:es.upv.dsic.gti_ia.jgomas.Cmanager(<number_of_agents>, <map>, <refresh_time>,<duration>)”
```

  Ejemplo:

```
java –classpath "lib\jade.jar;lib\jadeTools.jar;lib
\Base64.jar;lib\http.jar;lib\iiop.jar;lib
\beangenerator.jar;lib\jgomas.jar;student.jar;lib
\jason.jar;lib\JasonJGomas.jar;classes;.” jade.Boot -gui
“Manager:es.upv.dsic.gti_ia.jgomas.Cmanager(4, map_04, 125,10)”
```

  La última línea (en naranja) representa la llamada al agente mánager, con el nombre `Manager`, la clase `es.upv.dsic.gti_ia.jgomas.Cmanager` y los siguientes parámetros:

  - 4: número de agentes
  - Map_04: nombre del mapa
  - 125: tiempo de refresco en milisegundos
  - 10: duración de la ejecución en minutos

- Lanzar los agentes: Ejecutar el siguiente comando o el fichero `jgomas_launcher.bat` (Windows) o `jgomas_launcher.bat` (Linux):

```
java -classpath "lib\jade.jar;lib\jadeTools.jar;lib\Base64.jar;lib
\http.jar;lib\iiop.jar;lib\beangenerator.jar;lib
\jgomas.jar;student.jar;lib\jason.jar;lib\JasonJGomas.jar;classes;."
jade.Boot -container -host localhost
"<lista_de_agentes>”
```

  Ejemplo:

```
java -classpath "lib\jade.jar;lib\jadeTools.jar;lib\Base64.jar;lib
\http.jar;lib\iiop.jar;lib\beangenerator.jar;lib
\jgomas.jar;student.jar;lib\jason.jar;lib\JasonJGomas.jar;classes;."
jade.Boot -container -host localhost
"T1:es.upv.dsic.gti_ia.JasonJGomas.BasicTroopJasonArch(jasonAgent_AXIS.asl);T2:es.upv.dsic.gti_ia.JasonJGomas.BasicTroopJasonArch(jasonAgent_AXIS_MEDIC.asl);A1:es.upv.dsic.gti_ia.JasonJGomas.BasicTroopJasonArch(jasonAgent_ALLIED_FIELDOPS.asl);A2:es.upv.dsic.gti_ia.JasonJGomas.BasicTroopJasonArch(jasonAgent_ALLIED.asl)”
```

  Como en el caso del mánager, las últimas líneas se usan para definir los agentes. En este caso, se definen los agentes T1, T2, A1, y A2. Como ejemplo, en el caso del agente T1, su nombre se representa con T1, su clase con `es.upv.dsic.gti_ia.JasonJGomas.BasicTroopJasonArch` y finalmente se especifica el fichero .asl donde se encuentra su código, en este caso `jasonAgent_AXIS.asl`.

- Finalmente, para visualizar el motor gráfico se ejecutará el fichero run_jgomasrender.bat, que tiene el siguiente contenido:

```
set OSG_FILE_PATH=../../../../data
JGOMAS_Render.exe --server <hostname> --port <integer>
```

9. Anexo: Creencias, acciones y objetivos/planes disponibles en la aplicación de Jason-JGomas

9.1. Beliefs disponibles

`tasks([])`
Contiene la lista de tareas activas del agente. Ej:
`tasks([task(1000,"TASK_GET_OBJECTIVE","Manager",pos(224,0,224),""),task(1001,"TASK_WALKING_PATH","A2",pos(204,0,228),"")])`
En este ejemplo la lista tiene dos tareas activas:

- `task(1000,"TASK_GET_OBJECTIVE","Manager",pos(224,0,224),"")` que indica que debe ir a por el objetivo (la bandera) que está en la posición pos(224,0,224). La prioridad asignada a esta tarea es 1000
- `task(2000,"TASK_GIVE_MEDIPAKS","A2",pos(204,0,228),"")` que indica que debe ir a la posición pos(204,0,228) donde está el agente A2 esperando paquetes de medicina (se supone que el agente que tiene esta tarea activa es médico). La prioridad asignada a esta tarea es 2000

`task_priority(Task, Y)`
Sirve para indicar la prioridad de cada tarea para el agente. Estos valores pueden ser modificados en función de lo que decida el agente. Ej: `task_priority("TASK_GET_OBJECTIVE",1000)` indica que la prioridad de la tarea "`TASK_GET_OBJECTIVE`" es 1000.

`manager("Manager")`
Indica el nombre del agente Manager. En principio no debe tocarse

`team(X)`
Creencia que indica a que equipo pertenece el agente: “AXIS” o “ALLIED”

`type(X)`
Indica el tipo de soldado que es el agente: "CLASS_SOLDIER", "CLASS_MEDIC" o "CLASS_FIELDOPS"

`patrollingRadius(X)`
Sólo para agentes del equipo “AXIS”. Sirve para indicar el radio utilizado en tareas de patrolling. Inicialmente es 64. A mayor valor los agentes se alejaran más del objetivo a cuidar.

`fovObjects([])`
Contiene la lista de objetos que ahora mismo ve el agente. La estructura de un objeto es `[#, TEAM, TYPE, ANGLE, DISTANCE, HEALTH, POSITION].`
Ejemplo: ?fovObjects(Objects); guarda en la variable Objects la lista de objetos que ve el agente. Cada objeto de la lista tiene la estructura comentada.
Ejemplo de objeto en la lista: [1,200,1,0.58,14.76,78,pos(214,0,219)], el objeto 1 es del equipo 200 (AXIS), del tipo 1 (agente), con el ángulo 0,58, a una distancia de 14,76, que tiene una salud de 78 y su posición es pos(214,0,219)
Nota: Los valores de TEAM son: 100 (Allied), 200 (Axis), 0 (bandera)

`aimed("false")`
Indica que el agente no tiene un enemigo al que disparar.

`aimed("true")`
Indica que el agente tiene un enemigo al que disparar.

`medicAction(on)`
Indica que el agente (debe ser médico) está decidido a ayudar

`medicAction(off)`
Indica que el agente (debe ser médico) no está decidido a ayudar

`fieldopsAction(on)`

Indica que el agente (debe ser fieldops) está decidido a ayudar

`fieldopsAction(off)`
Indica que el agente (debe ser fieldops) no está decidido a ayudar

`objectivePackTaken(on)`
Indica que el agente ha cogido la bandera

`state(Estado)`
Esta creencia se utiliza para indicar el estado del agente en su máquina de estados: standing, eligiendo que tarea hacer o esperando; go_to_target, acudiendo a su próximo objetivo; target_reached, ha llegado al objetivo; quit, debe terminar.

`current_task(Task)`
Guarda la tarea actual. Ej: `current_task(task(1000,"TASK_GET_OBJECTIVE","Manager",pos(224,0,224),""))` la tarea actual es la de ir a por la bandera a la posición 222,0,224

`my_health(X)`
Guarda la salud del agente. El valor inicial y máximo es 100, cuando llega a 0, el agente se muere.

`my_ammo(X)`
Guarda la cantidad de balas que dispone el agente. El valor inicial es 100.

`my_position(X,Y,Z);`
Guarda la posición última conocida por el agente

`objective(ObjectiveX, ObjectiveY, ObjectiveZ)`
Guarda la posición última conocida del objetivo del agente

`my_ammo_threshold(X)`
Guarda el umbral de balas por debajo del cual el agente puede pedir ayuda o tomar una decisión. Inicialmente es 50

`my_health_threshold(X)`
Guarda el umbral de salud por debajo del cual el agente puede pedir ayuda o tomar una decisión. Inicialmente es 50

`debug(X)`
Indica el grado de verbosidad del agente. Entre 1 y 3

9.2. Acciones

`update_destination(Destination)`
Esta acción sirve para modificar el destino al que debería dirigirse el agente una vez calculado un nuevo posible destino. Por ejemplo cuando se decide ir a otro sitio para perseguir a un enemigo.

`move(Time);`
Esta acción es para indicar la acción de movimiento durante un tiempo Time. Su uso es interno, no es necesario usarla.

`check_position(Position)`
Sirve para saber si una posible posición es válida o no. Tras su ejecución se genera una creencia: position(X) donde X puede ser valid o invalid.

`create_medic_pack`
Esta acción es usada por los soldados médicos para generar paquetes de medicina. Su uso es interno, no es necesario usarla.

`create_ammo_pack`
Esta acción es usada por las soldados fieldops para generar paquetes de armamento. Su uso es interno, no es necesario usarla.

9.3. Acciones Internas añadidas

`.my_team(type,list)`
Esta acción devuelve la lista de agentes de tu equipo disponibles a través de las páginas amarillas (DF) que son del tipo "type". Ej: `.my_team("medic_AXIS", E)` ejecutado por un agente del equipo "AXIS" devuelve la lista E que contiene los médicos de mi equipo.
Los servicios disponibles por defecto son:

- "AXIS", ofrecido por todos los agentes del equipo AXIS
- "ALLIED", ofrecido por todos los agentes del equipo ALLIED
- "medic_AXIS", ofrecido por todos los agentes médicos del equipo AXIS
- "fieldops_AXIS", ofrecido por todos los agentes fieldops del equipo AXIS
- "backup_AXIS", ofrecido por todos los agentes soldier del equipo AXIS
- "medic_ALLIED", ofrecido por todos los agentes médicos del equipo ALLIED
- "fieldops_ALLIED", ofrecido por todos los agentes fieldops del equipo ALLIED
- "backup_ ALLIED ", ofrecido por todos los agentes soldier del equipo ALLIED

`.register( "JGOMAS", "type")`
Se utiliza para registrar en el DF un servicio del tipo "type" por parte del agente que lo ejecuta. Ej: `.register( "JGOMAS", "medic_AXIS")` registra en el DF el servicio "medic_AXIS".

`.send_msg_with_conversation_id(Rec, Perf, Cont, ConvId)`
Se utiliza para enviar un mensaje con id de conversación (es necesario por compatibilidad con el manager y con el resto de agentes). Ej: `.send_msg_with_conversation_id(ag1, tell, Content, "CFA")` envía un mensaje al agente ag1, con la performativa "tell", con el contenido Content y con el id de conversación "CFA" que significa Call for Ammo.

9.4. Objetivos (Objetivos y planes disponibles que pueden ser usados pero no modificados)

!look

Este objetivo se lanza para que el agente actualice los objetos que tiene a su alrededor. Actualiza fovObjects(ObjList), que puede ser usado para ver qué objetos hay cerca.
Su uso es interno, no es necesario usarla.

!shot(0)

Este objetivo se lanza cuando el agente decide disparar

!add_task(task(TaskPriority, TaskType, Agent, Position, Content))
!add_task(task(TaskType, Agent, Position, Content))

Estos dos objetivos se lanzan para añadir una nueva tarea en la lista de tareas del agente. TaskPriority: Prioridad de la tarea; TaskType: Tipo de tarea; Agent: agente asociado a la tarea; Position: Posición donde llevar a cabo la tarea; Content: Posible contenido adicional
Ejemplo: !add_task(task(500, "TASK_PATROLLING", M, pos(X, Y, Z), "")); añade una nueva tarea de tipo TASK_PATROLLING para ir a la posición (X,Y,Z) por parte del agente M

!safe_pos(x, y, z);

Este objetivo se lanza para calcular una posición valida cerca de la expresada mediante (x,y,z). Su ejecución crea una creencia nueva que indica dicha posición válida (safe_pos(X, Y, Z))
Ejemplo: Queremos saber una posición válida para ir lo más cerca posible de pos(300, 0, 30)

```
!safe_pos( 300, 0, 30 );
?safe_pos( X, Y, Z );
.println( "Es seguro ir a la posicion pos( ", X, ", ", Y, ", ", Z, " )" );
```

!distance(pos(A, B, C), pos (X, Y, Z));

Este objetivo calcula la distancia euclidea entre dos posiciones. El resultado se alamcena en una nueva creencia distance(D)
Ejemplo: !distance(pos(1, 0, 1), pos (4, 0, 3));

```
?distance( D );
.println( "La distancia entre los dos puntos es: ", D );
```

!nearest(Agents, K);
!nearest(Agents);

Este objetivo clacula, dada una lista de objetos/agentes, el más cercano ó el k-ésimo más cercano. Crea una creencia con el nombre, la posición y la distancia: nearest(Agent, Position, Distance)
Ejemplo: ?fovObjects(Agents);

```
.length(Agents, Length);
if (Length > 0) {
        !nearest( Agents );
        ?nearest( Agent, Position, Distance );
        .println( "El agente más cercano es ", Agent, " y está en la pos ", Position, " (distancia: ", Distance, ")" );
}
```

9.5. Objetivos (Objetivos y planes disponibles que pueden ser modificados)

!init

Este objetivo se lanza una sola vez en la inicialización del agente. Su plan asociado puede ser utilizado para introducir creencias o nuevos objetivos necesarios por la estrategia diseñada.

!update_targets

Este objetivo puede utilizarse para actualizar las tareas y sus prioridades. Se invoca cuando el agente pasa a estado standing y debe elegir nueva tarea entre las disponibles. Sería necesario implementar el plan asociado a la creación de este evento

`!perform_look_action`

Este objetivo se invoca cuando se ha mirado alrededor y se supone que se ha actualizado la lista de objetos alrededor `fovObjects(L)`. Sería necesario implementar el plan asociado a la creación de este evento para poder ver qué hay alrededor

`!get_agent_to_aim`

Este objetivo se invoca después del `!perform_look_action`, se utilizaría para decidir si hay algún enemigo al que apuntar. Una implementación sencilla del plan asociado ya está disponible. Sería interesante mejorar dicho plan asociado para tomar una decisión de a quien apuntar más refinada.

`!perform_aim_action`

Si hay un enemigo al que apuntar se lanza este objetivo, el cual puede servir para tomar alguna decisión respecto a qué hacer, ir a por él, pasar de él… Una implementación sencilla del plan asociado ya está disponible. Sería interesante mejorar dicho plan asociado para tomar una decisión de a quien apuntar más refinada.

`!perform_injury_action`

Este objetivo se lanza cuando el agente es disparado. Sería necesario implementar el plan asociado a la creación de este evento para tomar una decisión, por ejemplo huir si hay poca vida.

`!perform_no_ammo_action`

Este objetivo se lanza cuando el agente dispara y no le quedan balas. Sería necesario implementar el plan asociado a la creación de este evento para tomar una decisión, por ejemplo huir.

`!performThresholdAction`

Este objetivo se lanza cuando el agente dispone de menos vida o balas que los umbrales definidos en `my_ammo_threshold(X)` y en `my_health_threshold(X)`. Una implementación sencilla del plan asociado ya está disponible, la cual siempre pide ayuda a médicos o a fieldops de su equipo. Sería interesante mejorar dicho plan asociado para tomar una decisión más refinada de qué hacer.

`!checkMedicAction` (Sólo para médicos)

Este objetivo es lanzado cuando a un agente médico le llega una petición de ayuda. Una implementación sencilla del plan asociado ya está disponible, la cual siempre decide ayudar, para ello genera la creencia `medicAction(on)`. Sería interesante mejorar dicho plan asociado para tomar una decisión más refinada de qué hacer y en ocasiones generar la creencia `medicAction(off)`, la cual haría que el agente no atendiese la petición porque no le interesa.

`!checkAmmoAction (Sólo para fieldops)`

Este objetivo es lanzado cuando a un agente fieldops le llega una petición de ayuda. Una implementación sencilla del plan asociado ya está disponible, la cual siempre decide ayudar, para ello genera la creencia `fieldopsAction(on)`. Sería interesante mejorar dicho plan asociado para tomar una decisión más refinada de qué hacer y en ocasiones generar la creencia `fieldopsAction(off)`, la cual haría que el agente no atendiese la petición porque no le interesa.

`!setup_priorities`

Este objetivo se lanza en la inicialización del agente para fijar las prioridades de las tareas del agente. Cada agente puede tener sus propias prioridades. Una implementación sencilla del plan asociado ya está disponible. Sería interesante en ocasiones modificarlo para añadir nuevas tareas o cambiar las prioridades para tener agentes que se comportan de forma distinta.

`+cfm_agree[source(M)]`

Este objetivo se lanza cuando se recibe un mensaje confirmando que nos van a prestar ayuda médica. El agente M es el que atenderá la petición. No es necesario modificar el plan asociado aunque puede ser interesante en ocasiones modificarlo para refinar el comportamiento del agente.

`+cfm_refuse[source(M)]`

Este objetivo se lanza cuando se recibe un mensaje denegando la ayuda médica. El agente M es el que ha denegado la petición. No es necesario modificar el plan asociado aunque puede ser interesante en ocasiones modificarlo para refinar el comportamiento del agente cuando se le niega la ayuda.

`+cfa_agree[source(M)]`

Este objetivo se lanza cuando se recibe un mensaje confirmando que nos van a prestar ayuda de armamento. El agente M es el que atenderá la petición. No es necesario modificar el plan asociado aunque puede ser interesante en ocasiones modificarlo para refinar el comportamiento del agente.

`+cfa_refuse[source(M)]`

Este objetivo se lanza cuando se recibe un mensaje denegando la ayuda de armamento. El agente M es el que ha denegado la petición. No es necesario modificar el plan asociado aunque puede ser interesante en ocasiones modificarlo para refinar el comportamiento del agente cuando se le niega la ayuda.