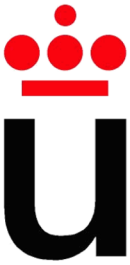


UNIVERSIDAD REY JUAN CARLOS

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERÍA INFORMÁTICA

Curso Académico 2009/2010

Proyecto de Fin de Carrera

mHealth:
Demostrador de emergencias médicas. Proyecto Tecnologías del Acuerdo

Autores:

Eva Rodríguez Yagüe
Miguel Isidoro Sánchez Gómez

Tutores:

Holger Billhardt
Roberto Centeno

mHealth:
Demostrador de emergencias médicas.
Proyecto Tecnologías del Acuerdo

Eva Rodríguez Yagüe

Miguel Isidoro Sánchez Gómez

Enero 2010

Resumen

Este trabajo fin de carrera se enmarca dentro del proyecto *Agreement Technologies*, en el que colabora el grupo de Inteligencia Artificial de la Universidad Rey Juan Carlos junto con otras instituciones. El objetivo del proyecto es el diseño e implementación de un demostrador de emergencias médicas llamado *mHealth*, que sirva para la evaluación y demostración de los resultados de investigación -algoritmos, técnicas y modelos- desarrollados en el proyecto AT.

Dicho demostrador trabajará en el dominio de las emergencias médicas, centrándose, entre todos los escenarios posibles, en el relacionado con el transporte de pacientes a hospitales que requieren asistencia. Como referencia se ha usado el modelo implementado en la Comunidad de Madrid a través del centro coordinador SUMMA112, desplegando en el demostrador los protocolos y normas que regulan su actividad.

Para el diseño e implementación del demostrador se utilizará un paradigma basado en agentes, donde cada entidad participante en el escenario de las emergencias médicas será representado por un agente software. Estos agentes interactuarán, siguiendo las normas y protocolos descritos por el centro coordinador, a través del entorno en el que se encuentran. Este entorno será simulado, consiguiendo así recrear un escenario lo más realista posible.

Toda la actividad llevada a cabo por los agentes en el entorno simulado podrá ser visualizada mediante una interfaz que permita observar sus características, tanto físicas -posiciones- como computacionales -interacciones.

Para el desarrollo del proyecto se utilizará la plataforma de agentes JADE, que será integrada con otras tecnologías como XML, Google Maps y JavaScript.

Índice general

Índice de figuras

Capítulo 1 Introducción

En este capítulo, se explican una serie de conceptos necesarios a modo de introducción para la comprensión del trabajo realizado en este proyecto. Muchos de esos conceptos se irán desarrollando con más detalle a lo largo de la memoria.

1.1. Introducción al proyecto AT

El trabajo realizado en este proyecto fin de carrera forma parte del proyecto de investigación *Agreement Technologies*, financiado por el Ministerio de Ciencia e Innovación, coordinado por el Instituto de Investigación en Inteligencia Artificial (IIIA-CSIC) y cuyos miembros colaboradores son la Universidad Politécnica de Valencia y la Universidad Rey Juan Carlos.

El principal objetivo del proyecto AT es el desarrollo de modelos, métodos y algoritmos para la construcción de un nuevo paradigma de comunicación en sistemas distribuidos. Este nuevo paradigma se estructura a través del concepto de acuerdo entre agentes software, el cual permite que los agentes, una vez aceptado el acuerdo, se requieran mutuamente para usar servicios.

Las líneas de investigación del proyecto AT se han estructurado en una arquitectura de torre de 5 niveles 1.1:

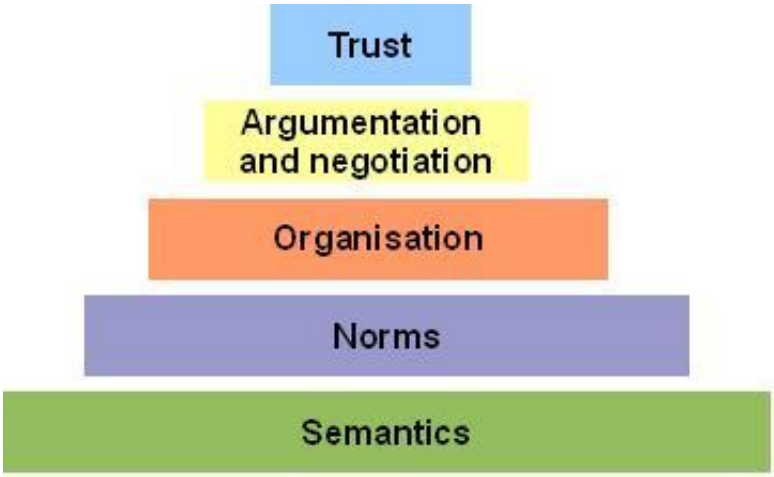


Figura 1.1: Arquitectura por niveles del proyecto AT

- Nivel 1: Semántica y gestión de recursos. Solucionar incompatibilidades semánticas y establecer ontologías para comprender las normas o acuerdos.
- Nivel 2: Definición de las normas que determinan las restricciones que todo acuerdo debe satisfacer. En este nivel se debe estudiar la capacidad del software para adaptarse a la normativa.
- Nivel 3: Organizar las estructuras sociales de los agentes: roles y relaciones entre ellos.
- Nivel 4: Este nivel es para el estudio de métodos de argumentación y negociación. Esto se traduce en alcanzar acuerdos que respeten las restricciones que las normas y organizaciones imponen a los agentes.
- Nivel 5: La capa de confianza que se encargará de construir las relaciones entre agentes a partir de la reputación que adquieran estos. Muy importante para un sistema abierto como éste.

Los avances logrados en el aspecto teórico del proyecto AT son probados a través de tres demostradores software: *eProcurement* (comercio electrónico),

mHealth (servicios sanitarios móviles) y *mWater* (gestión de recursos hídricos). El segundo de ellos, el demostrador *mHealth*, forma parte del desarrollo de este proyecto fin de carrera. *mHealth* engloba aquellos escenarios donde la gente que, por ejemplo, está de viaje en un país extranjero, cae enferma y requiere una asistencia medica a la que desconoce como acceder. El demostrador debe ayudar a esa gente a acceder rápidamente al servicio de emergencias médicas y también ayudar a los profesionales sanitarios a obtener la información del paciente que consideren importante de forma sencilla y rápida, para mejorar su asistencia. Con este demostrador se pretende, por un lado poder utilizar y evaluar los resultados de investigaciones teóricas en escenarios reales y por otro, construir una aplicación prototipo que simule problemas reales dentro del dominio de las emergencias médicas.

1.2. Introducción a las emergencias médicas

Para el desarrollo del demostrador *mHealth*, se está colaborando con instituciones como el SUMMA112 de la Comunidad de Madrid y el Hospital de Fuenlabrada. Ambas proporcionan el conocimiento y la información que se necesita sobre el dominio de las emergencias médicas, conocimiento muy importante para lograr un demostrador lo más real posible.

Con ayuda del SUMMA112, se identificaron tres posibles escenarios muy habituales en el dominio de las emergencias médicas:

- Transporte de urgencias: este caso comienza cuando una persona requiere asistencia médica de urgencias y contacta con el centro coordinador de emergencias. Una ambulancia se envía hacia el paciente para su asistencia y si se requiere, se le transporta hacia un hospital para que le atiendan.
- Coordinación de transferencias interhospitalarias: el centro coordina los

traslados de pacientes desde un hospital a otro cuando, por ejemplo, el hospital donde está ingresado no tiene los recursos necesarios para llevar a cabo una operación concreta.

- Asistencia telefónica: este escenario, similar al primero, comienza cuando un paciente llama a un centro de emergencias solicitando asistencia. Los médicos que se encuentran en el centro telefónico determinan que no es necesario el transporte del paciente. En ese caso, el paciente puede ser atendido telefónicamente y además, hacer uso de servicios disponibles para ayudarle a encontrar un ambulatorio cercano, una farmacia de guardia, o cualquier tipo de información que se considere útil en este tipo de escenarios.

En cualquiera de estos tres casos, es un centro de emergencias el que coordina los recursos necesarios, como son las ambulancias, hospitales, equipo médico, etc. Diferentes centros de emergencias tienen diferentes formas de manejar sus recursos. En este demostrador se utiliza el protocolo que sigue el SUMMA112 en la Comunidad de Madrid y, en concreto, se utiliza el primer escenario como ejemplo de un caso típico de emergencia médica.

El modelo de actuación del SUMMA112 en cuanto al transporte de urgencias, se basa en coordinar el transporte usando un conjunto de normas y protocolos que indican cómo los diferentes actores del escenario deben tomar sus decisiones. El flujo básico de este escenario consiste en:

1. El paciente solicita asistencia: una persona empieza a encontrarse mal y elige el centro de emergencias al que va a solicitar asistencia. En este caso, siempre llama al SUMMA112 por ser el único centro en Madrid.

2. El centro coordinador envía una ambulancia hacia el paciente: cuando el centro recibe la llamada, un operador telefónico obtiene los datos de la persona que llama y se los comunica a un médico que se encuentra en

el centro para que evalúe la gravedad del paciente y seleccione el tipo de recurso que debe usarse en la asistencia del paciente. Una vez que se sabe el tipo de recurso, se selecciona una ambulancia que cumpla esos requisitos y que, además, sea idónea para esa asistencia, ya sea por su localización, estado, equipo médico, etc.

3. La ambulancia asiste al paciente: La ambulancia se mueve hacia el paciente, decidiendo previamente qué camino seguir para llegar lo antes posible y, una vez allí, realiza una primera asistencia *in situ* al paciente. Basándose en el estado del paciente, el equipo médico de la ambulancia determina si debe transportar al paciente a un hospital o no.

4. La ambulancia transporta al paciente hacia un hospital: si decide transportarlo, debe elegir el hospital más adecuado para la atención al paciente, ya sea por su localización, instalaciones de urgencias o disponibilidad. Una vez seleccionado el hospital, se elige la mejor ruta hasta él y se traslada al paciente.

5. El hospital atiende al paciente: cuando la ambulancia llega al hospital, éste se hace cargo del paciente, ingresándole y asistiéndole para conseguir curarle.

1.3. Agentes software

Existen múltiples definiciones del término agente, una de las mas citadas [**?**] es la siguiente: "un agente es un sistema informático situado en un entorno y que es capaz de realizar acciones de forma autónoma para conseguir sus objetivos de diseño". Para conseguir dichos objetivos se tienen en cuenta las siguientes características:

- Reactividad: los agentes mantienen una interacción constante con su entorno y responden a los cambios que ocurren en él en un tiempo finito.
- Pro-actividad: además de reaccionar ante el entorno los agentes deben ser capaces de tomar la iniciativa para generar y tratar de alcanzar sus metas.
- Habilidad social: para lograr sus metas, los agentes pueden interactuar con otros agentes por medio de algún tipo de lenguaje de comunicación (por ej. FIPA-ACL [**?**]). Estas comunicaciones se realizarán mediante el envío de conceptos los cuales estarán definidos utilizando un vocabulario común entre los agentes, a este vocabulario se le denomina ontología, ver 1.4.1.

En nuestro proyecto seguimos la visión de que el entorno de un agente es todo aquello que le rodea. Por tanto, todas las acciones realizadas por éstos se llevarán a cabo por medio del entorno, produciendo cambios en éste. El nuevo estado del entorno será percibido por los agentes, que elegirán la siguiente acción en base a estas percepciones.

1.4. Sistemas multiagente

Podemos definir un sistema multiagente como un *sistema compuesto por un conjunto de agentes autónomos, generalmente heterogéneos y potencialmente independientes, que interaccionan (se comunican) entre sí para alcanzar una serie de objetivos (el tipo de interacción puede ser cooperativa o competitiva).* En general, un sistema multiagente presentará las siguientes características:

- Cada agente tiene un conocimiento limitado. Esta limitación puede ser tanto del conocimiento del entorno, como de las intenciones de los demás agentes a la hora de realizar sus propias tareas.

- No hay un sistema de control global, lo cual permite la interconexión e interoperabilidad entre diferentes sistemas.
- Los datos están descentralizados para que los agentes puedan actuar de forma autónoma.
- La computación es asíncrona.

La creación de un sistema multiagente permite la gestión inteligente de un sistema complejo, coordinando los distintos subsistemas que lo componen e integrando los objetivos particulares de cada subsistema en un objetivo común. La colaboración entre los componentes del sistema permitirá la resolución del problema y la consecución de los objetivos predefinidos. Estos componentes serán agentes software los cuales podrán realizar determinadas tareas y se coordinarán con el resto de agentes mediante un proceso de comunicación, de manera que el razonamiento interno de un agente consistirá en la toma de decisiones y en la identificación de la información que se debe compartir.

Un mecanismo para coordinar agentes heterogéneos en entornos abiertos son las organizaciones virtuales. Este tipo de organizaciones tiene la característica de utilizar conceptos sociales tales como roles, normas, etc. De esta manera los agentes pertenecientes a una misma organización se encargarán de coordinar los recursos y servicios propios a la organización.

1.4.1. Ontología

El propósito de las ontologías es dar significado a los conceptos que manejan los agentes, dando lugar a que el contenido de los mensajes que se intercambien los agentes sea comprendido por todos aquellos que compartan la misma ontología. De esta manera los agentes serán capaces de compartir conocimiento.

Según la definición literal de FIPA [**?**], la ontología es un conjunto de símbolos con una interpretación asociada que puede ser compartida por una comunidad de agentes. La ontología incluye un vocabulario de símbolos referidos a los objetos de un dominio concreto, o bien símbolos referidos a relaciones que pueden darse en dicho dominio.

En un mensaje, la ontología proporciona el vocabulario a utilizar para poder interpretar el contenido del mensaje aportando, también, la semántica que especifica el dominio del conocimiento del que trata dicho mensaje.

Cuando se desea crear una ontología hay que diferenciar los siguientes elementos:

- Concepto: representa las entidades que forman parte de la ontología. Por ejemplo ambulancia, paciente, etc.
- Predicado: se utilizan para relacionar conceptos ya que un concepto por sí solo no tiene significado.
- Acción: son acciones que pueden realizar los agentes.

Una vez definidos los elementos de la ontología los agentes podrán utilizar este vocabulario común para poder compartir conocimiento.

1.4.2. Estándar FIPA

En el estándar FIPA [**?**] (Foundation for Intelligent Physical Agents) se definen las características que deben cumplir las plataformas de gestión de sistemas multiagentes de tal manera que se asegura la interoperabilidad entre sistemas heterogéneos. Para ello el modelo FIPA establece el modelo lógico referente a la creación, destrucción, registro, localización y comunicación de agentes. Algunas de las implementaciones basadas en la especificación FIPA son FIPA OS y JADE, esta última implementación será la que utilizaremos en nuestro proyecto.

Para establecer este modelo lógico, FIPA establece una arquitectura abstracta, ver 1.2, compuesta por los siguientes elementos:

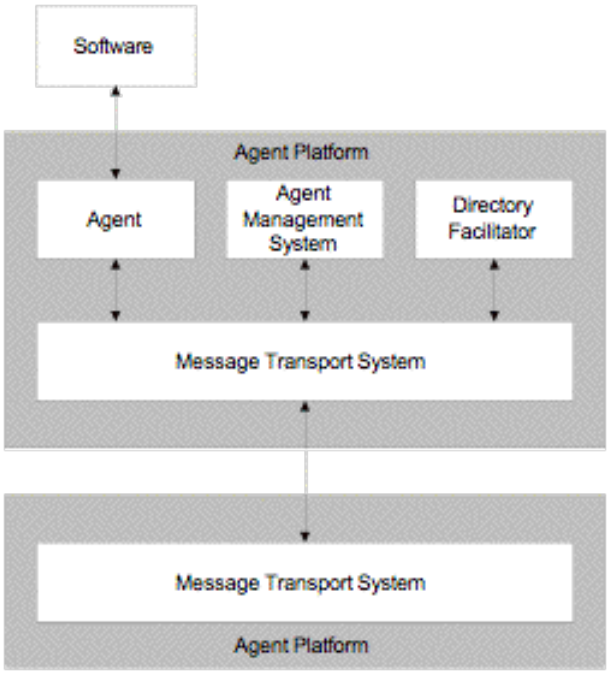


Figura 1.2: Modelo de referencia de FIPA

- *Agent*: es un proceso computacional que implementa la autonomía y la funcionalidad de comunicación de una aplicación. Los agentes pueden acceder a la plataforma de agentes registrándose en el *Agent Management System* y además pueden registrar sus servicios en el *Directory Facilitator*, tendrán asociado un identificador único (AID) y se podrán comunicar con otros agentes mediante mensajes ACL, ver 1.4.2.1.

- *Directory Facilitator (DF)*: es un componente opcional que proporciona un servicio de páginas amarillas en la que los agentes pueden registrarse junto con los servicios que ofrecen, el lenguaje que utilizan, y las ontologías que pueden usar. De esta manera el resto de agentes pueden consultar este DF para encontrar un agente que ofrezca un servicio concreto.

- *Agent Management System (AMS)*: es un componente requerido que supervisa el acceso y uso de la plataforma de agentes. Solo puede existir un AMS en cada plataforma de agentes. Proporciona un servicio de páginas blancas en el que los agentes se registran para obtener un AID válido y en el cual se encuentran las direcciones asociadas a todos los agentes existentes en la plataforma.

- *Message Transport Service (MTS))*: es el elemento encargado de gestionar el envío de mensajes entre agentes de una plataforma y entre agentes de distintas plataformas.

- *Agent Platform (AP)*: es el núcleo del modelo de referencia de FIPA y se encarga de proporcionar la infraestructura para el desarrollo y uso de agentes. Se compone del hardware en el cual se ejecuta el software, el sistema operativo, el software que da soporte a los agentes, y los componentes de administración de agentes (*directory facilitator*, *agent management system* y *message transport service*). Cabe destacar que el concepto de plataforma de agentes no implica que todos los agentes deban residir en el mismo ordenador, por ello el estándar FIPA permite diferentes implementaciones de una plataforma de agentes siempre y cuando posean un mecanismo adecuado de interoperabilidad, es decir, FIPA sólo se centra en la comunicación de los agentes dentro de la propia plataforma y con los agentes que residen en otras plataformas.

1.4.2.1. Agent Communication Language (ACL)

ACL define la estructura que deben tener los mensajes que se envían los distintos agentes. Un mensaje FIPA ACL contiene información referente al acto comunicativo en el que se realizan, los distintos participantes de la comunicación,

el contenido del mensaje, la descripción del contenido y otra serie de datos para el control de la conversación entre los agentes, podemos ver desglosados estos elementos en la figura 1.3.

Parameter	**Category of Parameters**
performative	Type of communicative acts
sender	Participant in communication
receiver	Participant in communication
reply-to	Participant in communication
content	Content of message
language	Description of Content
encoding	Description of Content
ontology	Description of Content
protocol	Control of conversation
conversation-id	Control of conversation
reply-with	Control of conversation
in-reply-to	Control of conversation
reply-by	Control of conversation

Figura 1.3: Elementos de mensaje de FIPA ACL

El acto comunicativo entre agentes también puede ser denominado *performative*, un mensaje ACL debe tener obligatoriamente este campo ya que indica la intencionalidad del mensaje y los roles que desempeñarán los agentes implicados en la comunicación. Los diferentes tipos de performativas los podemos ver en la figura 1.4

Accept Proposal	Agree	Cancel	Call for Proposal
Confirm	Disconfirm	Failure	Inform
Inform If	Inform Ref	Not Understood	Propagate
Propose	Proxy	Query If	Query Ref
Refuse	Reject Proposal	Request	Request When
Request Whenever	Subscribe		

Figura 1.4: Repertorio de actos comunicativos

La comunicación entre agentes a menudo posee patrones típicos de secuencias de mensajes. A este tipo de patrones se les denomina *interaction protocols*. En la

especificación FIPA podemos encontrar algunos de los protocolos más comunes utilizados en las aplicaciones basadas en agentes.

Los protocolos de comunicación recogidos en el estándar FIPA son los siguientes:

- *Request*: un agente solicita a otro agente que realice cierta acción. Podemos ver el diagrama de este protocolo en la figura 1.5.
- *Request when*: un agente solicita a otro agente que realice cierta acción siempre que se cumpla una precondición.
- *Query*: un agente solicita a otro agente que informe sobre algo.
- *Contract net*: un agente solicita la realización de una tarea a un conjunto de agentes, estos dan su propuesta y el iniciador selecciona a uno de ellos para que realice la tarea.
- *Brokering*: un agente ofrece las funcionalidades de otros agentes o reenvía las peticiones al agente apropiado.
- *English auction*: los agentes participan en una subasta en la que el precio va subiendo progresivamente.
- *Dutch auction*: los agentes participan en una subasta en la que el precio va bajando progresivamente.
- *Recruiting*: similar al *brokering* pero las respuestas sobre el servicio van directamente al agente que lo necesita.
- *Propose*: un agente propone a una serie de agentes la realización de una tarea y estos aceptan o no.

- *Subscribe*: un agente pide ser notificado si cierta condición se vuelve verdadera.

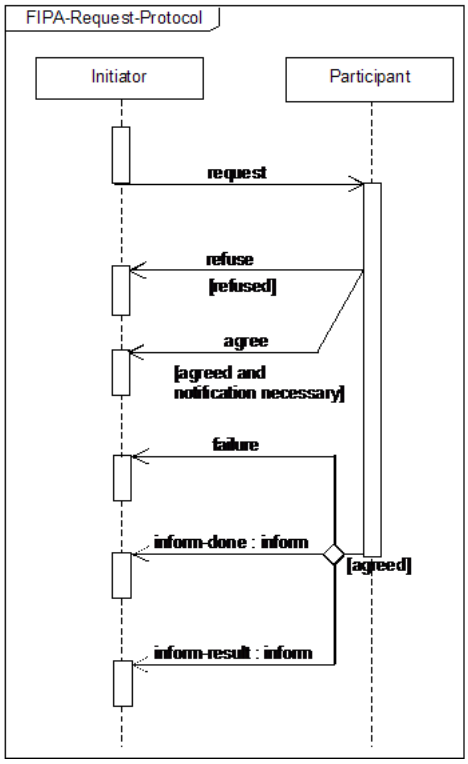


Figura 1.5: Diagrama de protocolo *request*

Para que los agentes de una aplicación sean capaces de comunicarse se necesita algo más que compartir el mismo lenguaje, en este caso FIPA ACL, ya que si los conceptos del contenido del mensaje no son utilizados en el mismo contexto pueden llegar a no entenderse. La solución a este problema es crear una ontología común en la que se definan los términos y relaciones básicas que componen el vocabulario de un determinado dominio.

1.4.3. JADE

JADE [**?**] (*Java Agent DEvelopment Framework*) es un framework para el desarrollo de agentes software en Java que cumple con las especificaciones del estándar FIPA [**?**].

En este framework podemos encontrar las librerías necesarias para la creación de agentes, los cuales podrán comunicarse entre sí utilizando el lenguaje FIPA ACL (ver 1.4.2.1). Además nos proporciona una serie de agentes ya implementados que implementan los elementos necesarios de la arquitectura abstracta del estándar FIPA, así como otras utilidades que facilitan el control de los agentes y sus interacciones. Los agentes que se encargan de estas nuevas funcionalidades son dos: el agente *Sniffer*, que se encarga de interceptar mensajes entre distintos agentes y mostrarlos de manera gráfica para facilitar la depuración; y el agente *Introspector*, que permite controlar el ciclo de vida de un agente en ejecución y los mensajes que intercambia.

Para establecer los distintos roles de los agentes, JADE nos ofrece comportamientos o *behaviours* en los que se podrá establecer el flujo de tareas que deberán realizar. Además, para facilitar la comunicación entre agentes, JADE implementa los protocolos estándar identificados en FIPA. La ontología común que utilicen los agentes para estas comunicaciones podrá ser representada mediante objetos Java de una manera sencilla.

Por último destacar que los agentes residirán en contenedores denominados *container*. Estos contenedores podrán ser distribuidos a partir de la misma plataforma de agentes, la cual contendrá el contenedor principal de la aplicación.

1.5. Introducción al concepto de simulador de entornos físicos

Tal y como se ha explicado en el apartado anterior, los sistemas multiagentes se componen de un conjunto de agentes inteligentes que se localizan y actúan en un entorno. Este entorno se refiere a un componente software que, junto con el sistema de agentes inteligentes, representa un mundo de interés, como puede ser el mundo de las emergencias médicas.

Los agentes modelan el comportamiento de los actores que forman parte de ese mundo de interés y normalmente, cada uno de ellos tiene sus propios objetivos y toma sus propias decisiones. Las restricciones y leyes que gobiernan el mundo de esos agentes se definen en el entorno donde actúan, de manera que dicho entorno debe representar la parte del mundo real que nos interesa de forma consistente y coherente.

Dado que existen ciertos escenarios del mundo real que son complejos de usar para experimentaciones y pruebas, en la práctica es muy común desarrollar aplicaciones que simulen este tipo de escenarios, las cuales facilitan el poder controlar todo lo que esta sucediendo en el entorno simulado, permiten repetir experimentos, introducir aleatoriedad, probar nuevas situaciones, elegir la simulación de casos concretos, etc. Además, simplifican la recogida de los datos obtenidos en los experimentos para su posterior estudio y todo esto, abaratando el coste y esfuerzo que conllevaría realizar las simulaciones en escenarios reales. Una simulación consiste en diseñar un modelo de un sistema real objeto de estudio y realizar experimentos con él, con el fin de comprender el comportamiento de dicho sistema o evaluar nuevas estrategias para su deseado funcionamiento. Cuando se utilizan sistemas multiagentes, la simulación consiste en hacer que el entorno reemplace las entidades del mundo de interés por un sistema de agentes con una equivalencia

virtual. Dicho simulador debe ajustarse al modelo real, de forma que los hechos que se sucedan en el entorno desarrollado, se correspondan con posibles hechos reales para dar por válido el comportamiento del sistema implementado.

El simulador de emergencias médicas de este proyecto es un sistema multiagente compuesto de un único entorno y varios agentes inteligentes que representan los cuatro posibles actores del escenario propuesto: ambulancias, hospitales, pacientes y centros coordinadores de emergencias. El entorno simulado debe incorporar toda la información necesaria para representar muchas instantáneas del entorno real. Cada una de esas instantáneas contiene un conjunto de atributos y propiedades del entorno, conocido como "estado interno". Los agentes del sistema, al modelar entidades del mundo real, también se pueden componer de estados internos que caracterizan al agente en cada instante de tiempo. Según sea el escenario elegido para la simulación, el estado interno de cada agente se compondrá de unas características u otras. En nuestro caso, el estado interno del entorno va a ser la unión de los estados internos de cada agente que opera en él.

En este tipo de simulaciones, un factor muy importante es el tiempo. Lo ideal es que una simulación tenga alta velocidad de cómputo, pero en nuestro proyecto más que tener una alta velocidad de procesamiento se busca tener una simulación lo más realista posible. En esta aplicación, el tiempo se ha modelado de forma que la simulación esté compuesta de un número de pasos denominados *timesteps*, y cada uno de esos pasos tenga una duración idéntica, determinada por el usuario. Cada *timestep* albergará una iteración del protocolo de comunicación que existe entre el entorno y los agentes (envío de acciones- recepción de percepciones), de manera que en cada *timestep*, se tiene un estado interno del entorno actualizado tras la simulación de las acciones requeridas por los agentes.

Capítulo 2 Objetivos

En este capítulo, primero se exponen los motivos que nos han llevado a realizar el proyecto que se presenta en esta memoria y a continuación, se enumeran los objetivos que se quieren conseguir y que nos darán una solución al problema planteado:

2.1. Descripción del problema

En la vida real existen situaciones en las que gente que, por ejemplo, se encuentra de vacaciones en un país extranjero o trabajando en una región distinta a la suya, cae enferma y no sabe a dónde acudir para solicitar asistencia de urgencias. El demostrador *mHealth* pretende ayudar a esa gente a localizar el centro de emergencias que le proporcionará asistencia médica. Además, el *mHealth* pretende ayudar al propio centro de emergencias a obtener la información del paciente que llama de forma más rápida y sencilla, así como mejorar la coordinación de recursos que lleva a cabo el centro de emergencias, ayudándole en su toma de decisiones.

Estas funcionalidades se pueden desarrollar utilizando diversas técnicas y siguiendo diferentes protocolos de actuación. Es por ello que el objetivo del demostrador *mHealth* no sólo consiste en implementar una aplicación en particular,

sino que pretende servir como evaluador de todas esas diferentes técnicas y modelos que se investigan en el marco del proyecto AT.

El escenario de emergencias médicas seleccionado en este proyecto se basa en el transporte de urgencias de pacientes que han solicitado asistencia médica a un centro de emergencias. En este escenario, surgen diversas situaciones en las que los diferentes actores que participan deben tomar decisiones, que serán unas u otras dependiendo del modelo de emergencias médicas que se implante en el demostrador. En base a esas decisiones, se cumplirán o no los objetivos marcados en ese escenario, como pueden ser: conseguir curar al paciente que solicitó la asistencia, ahorrar en costes de gasolina, usar el mínimo número de recursos, etc. Por ello, al usar el demostrador, se podrán obtener los datos de un escenario simulado y corregir aquellas técnicas empleadas en él que supongan no conseguir un resultado óptimo para el usuario.

En este dominio de emergencias médicas participan diversas entidades del mundo real que deben modelarse en nuestro sistema. Su modelización debe corresponderse lo mejor posible con la realidad y deben tener comportamientos coherentes. Las diferentes entidades que participan en este escenario son:

- Centro de emergencias: se deben modelar como un conjunto de operadores telefónicos que reciben las llamadas de los pacientes y las procesan. Sirven de mediadores entre los pacientes y las ambulancias y deben decidir qué ambulancias se asignan a qué pacientes.

- Pacientes: representan a las personas que requieren asistencia médica y llaman a un centro de emergencias solicitándolo. Deben informar al centro de sus síntomas y localización, entre otros datos, para que éste seleccione la ambulancia que le enviará. Los pacientes, al enfermar, pueden morirse o curarse si reciben la adecuada asistencia médica. El tiempo que tarda en morirse o la probabilidad de curación se determinan en base al nivel de

enfermedad que tiene el paciente.

- Ambulancias: son los vehículos que se usan para el transporte de pacientes. Existen ambulancias de diferentes niveles, según sea su equipación médica y en base a eso, se enviarán a unos pacientes o a otros. Los centros coordinadores son quienes avisan a las ambulancias sobre los pacientes que necesitan asistencia. Sus tareas principales son asistir pacientes y transportarles a hospitales. En esas tareas se deben decidir ciertos aspectos, como el tiempo de asistencia a un paciente en base a su capacidad, el hospital al que transportarán al paciente y la ruta que deben seguir tanto al ir hacia al paciente como al llevarlo a un hospital.

- Hospitales: son las entidades que tienen infraestructura para atender a los pacientes enfermos. Cada hospital tiene diferentes características, desde el número de trabajadores que forman parte de él hasta el número de camas que tiene. Un hospital decide si admitir o no a un paciente, ya sea por un transporte de una ambulancia o porque un paciente llega por su propio pié o desde un traslado interhospitalario. Estos dos últimos casos se corresponden con los otros dos escenarios presentados en la introducción de esta memoria y, aunque no forman parte de este escenario médico, se deben poder simular para dotar de mayor realismo a los hospitales que se simulen. El hospital atenderá a los pacientes y mejorarán su estado físico o lo empeorarán dependiendo del nivel de urgencias que tengan.

2.2. Objetivos del proyecto

Una vez descritas las características de las emergencias médicas que nos llevan a realizar este proyecto, pasamos a detallar los objetivos que se desean cumplir:

- Simulador: el objetivo principal es desarrollar un simulador que muestre las interacciones entre diversos agentes en un escenario de emergencias médicas.
  - Este simulador debe proveer un entorno donde las acciones de las diferentes entidades que forman el escenario de emergencias médicas se puedan ejecutar, analizar y evaluar.
  - El entorno se podrá configurar, para poder realizar diferentes experimentos y permitir describir escenarios concretos.
- Aplicación (agentes): desarrollar un sistema multiagente basado en el estándar FIPA donde cada agente tenga definido un rol de los que hay en un escenario de emergencia médica, se vea como actúan y la comunicación que existe entre ellos.
  - La aplicación interactuará con el entorno del simulador a través de un protocolo de acción-percepción, en el cual los agentes pedirán al entorno que simule una lista de acciones y una vez emuladas, el entorno contestará a cada agente con el nuevo estado del mundo tras la simulación de esas acciones.
  - La secuencia de decisiones que tomarán los agentes en cada simulación dependerá de las diferentes técnicas empleadas en la implementación de la aplicación.
- Visualizador: se deberá mostrar gráficamente el flujo que realizan los agentes de la aplicación en cada simulación.
  - Se debe tener una visión global de lo que está pasando en cada momento en el entorno que se simula.
  - Se podrá ver individualmente la información asociada a cada agente que participa en el entorno: sus estados y sus acciones.

Capítulo 3 Descripción Informática

Una vez explicados los conceptos más importantes para la comprensión del proyecto y detallados los objetivos que se pretenden conseguir con su implementación, se describe a lo largo de este capítulo el proceso de desarrollo del sistema creado en este trabajo final de carrera.

En primer lugar, se explica la metodología empleada para el desarrollo del demostrador *mHealth* y a continuación, se van describiendo con más detalle cada una de las fases de esta metodología, de forma que se obtengan diferentes vistas del proyecto implementado, desde un nivel más abstracto hasta un nivel más detallado, para finalmente exponer una demostración de la funcionalidad del sistema creado.

3.1. Metodología

Las metodologías de desarrollo ayudan a crear un sistema software siguiendo unas pautas para producir el software deseado. En este proyecto, se ha usado un modelo de proceso de desarrollo similar al *Modelo Incremental*. Este modelo combina elementos del modelo clásico secuencial, formado por las etapas básicas del ciclo de vida del software, con la filosofía iterativa de la construcción de prototipos. Su desarrollo consiste en la segmentación del proyecto completo en pequeños periodos de tiempo denominados iteraciones, donde cada iteración es una secuencia de las fases de análisis, diseño, implementación y pruebas. Al final de cada iteración se obtiene un prototipo del sistema a desarrollar, y si el prototipo necesita mejorarse o ampliarse, se planifica el siguiente incremento y se itera de nuevo hasta conseguir el producto final. Esta metodología presenta la ventaja de ser dinámica y flexible, consiguiendo una gran adaptabilidad en la introducción de nuevos requisitos y en la modificación de los ya existentes.

En cada fase de análisis, se realizaron diversas reuniones con los tutores del proyecto para realizar la captura de requisitos. Se concretaron los agentes que formarían parte de la plataforma junto con sus comportamientos, estudiando cuales debían usarse como organizaciones de agentes para emular organizaciones reales, y se detallaron los protocolos de comunicación que usarían entre ellos, definiendo qué acciones estaban permitidas para cada tipo de agente y qué debían percibir de su entorno en cada comunicación.

En la fase de diseño, primero se estudiaron las diferentes tecnologías que se iban a usar a través de manuales y libros, incluyendo la asistencia a un seminario de JADE impartido en la Universidad Politécnica de Madrid por Giovanni Caire y Giovanna Sacchi. Una vez estudiadas las tecnologías, se procedió al diseño de la funcionalidad del sistema basándonos en la etapa previa de análisis así como

a diseñar la interfaz de usuario.

En la etapa de implementación y pruebas, realizamos la codificación de la aplicación y los test necesarios para probar que el sistema funcionaba correctamente según los requisitos previos. Con los resultados de esas pruebas y con las nuevas funcionalidades solicitadas en las reuniones con los tutores, se tenía la información necesaria para proceder a iterar sobre el prototipo creado y mejorar la funcionalidad del demostrador, hasta llegar al sistema que se presenta en este proyecto fin de carrera.

También como parte de esta metodología de desarrollo, se ha usado el Lenguaje de Modelado Unificado (UML, *Unified Modeling Language*) para la construcción de los distintos diagramas y modelos presentes en esta memoria, los cuales muestran abstracciones del mundo real de forma gráfica y visual mediante un vocabulario y unas reglas proporcionadas por este lenguaje. En definitiva, UML sirve para definir un sistema, detallando sus componentes en las etapas de análisis, diseño e implementación, y documentar el proyecto realizado para mejorar su comprensión.

3.2. Requisitos

Un requisito es una descripción de cómo se debe comportar el sistema, indicando las propiedades que debe cumplir el sistema que se va a desarrollar. No forman parte de los requisitos los detalles de diseño, implementación o pruebas que se vayan a realizar, como tampoco lo es la planificación del proyecto y sus necesidades.

A continuación, se presentan tanto los requisitos funcionales como los no funcionales del sistema:

3.2.1. Requisitos funcionales

Los requisitos funcionales capturan el comportamiento del sistema. La enumeración de estos requisitos ha sido subdividida en tres subapartados: aplicación, simulador y visualizador. Cada apartado se corresponde con un subsistema del proyecto, y cada uno de ellos debe cumplir obligatoriamente las siguientes condiciones:

3.2.1.1. Aplicación

- Debe ser un sistema multiagente distribuido.
- Los agentes deben ser racionales, es decir, serán capaces de percibir el entorno, procesar las percepciones y actuar en consecuencia teniendo en cuenta su estado interno, sus percepciones y las acciones que puedan llevar a cabo.
- Será capaz de poder probar e implementar mecanismos organizativos sobre los agentes y organizaciones.
- Debe permitir incluir actores relacionados con el dominio de las emergencias médicas:

  - Pacientes: serán capaces de moverse y de llamar a un centro de emergencias.
  - Centros de emergencias: serán capaces de atender llamadas y gestionar la asignación de ambulancias a los diferentes pacientes.
  - Ambulancias: podrán moverse, acudir al lugar en el que se encuentra un paciente para atenderle *in situ*, o trasladar pacientes a un hospital.
  - Hospitales: serán capaces de atender a los pacientes que acudan a ellos.

- Los actores del dominio deben ser capaces de realizar acciones y comunicarse entre ellos.
- Los pacientes y las ambulancias podrán realizar una única acción o comunicación en cada *timestep*.
- Las organizaciones (centros coordinadores y hospitales) podrán realizar en un mismo *timestep* tantas acciones o comunicaciones como miembros posean.

3.2.1.2. Simulador

- Será capaz de simular un entorno con las siguientes características:
  - Simulación síncrona y con *timesteps* definidos.
  - El entorno tiene una visión del estado interno de cada agente que participa en él.
  - El entorno informará a cada agente lo que pueden percibir del entorno por cada *timestep*.
  - El entorno será capaz de recibir acciones de los agentes.

- Simular como afectan las variables del entorno a los actores del dominio:
  - Simular como enferman los pacientes.
  - Simular la ocupación de un hospital.
  - Simular el movimiento de las ambulancias.
- El entorno también podrá ser configurable, pudiendo especificar:
  - Número de pacientes que se generarán durante la simulación, incluyendo el rango de niveles de atención médica para la simulación.
  - Especificar el número de centros coordinadores que habrá, indicando el número de operadores telefónicos de cada uno.
  - Se podrán incluir ambulancias al entorno, de forma aleatoria, mediante un fichero de ambulancias o especificando uno a uno sus parámetros. Estos parámetros son, entre otros, la adecuación del equipo médico de cada ambulancia para cada cierto tipo de emergencia.
  - Se podrán incluir hospitales al entorno, de forma aleatoria, mediante un fichero de hospitales o especificando uno a uno sus parámetros. Estos parámetros son, entre otros, el nivel de emergencia que es capaz de resolver y cuántos miembros componen el área de urgencias.
- Se podrán almacenar datos relativos al entorno:
  - Almacenar datos de configuración.
  - Almacenar datos de simulación.

- Se podrá configurar la duración de la simulación.
- Se podrán almacenar los datos relativos a la simulación.

3.2.1.3. Visualizador

- Una simulación podrá ser visualizada en tiempo real o guardada para posteriormente visualizarla si se desea.
- El visualizador permitirá ver de manera global el comportamiento de los agentes del sistema.
- El visualizador mostrará una lista de los agentes para facilitar el acceso a la información individual de cada agente.

3.2.2. Requisitos no funcionales

Los requisitos no funcionales son aquellos que no han sido especificados por el cliente, pero que se espera que el sistema recoja:

- La interfaz del programa debe ser intuitiva y de fácil utilización.
- La distribución de los componentes visuales en la interfaz debe permitir una cómoda visualización de la información que se quiere mostrar al usuario.
- El sistema debe responder adecuadamente ante cualquier imprevisto durante su ejecución.

3.2.3. Desarrollo de objetivos

El proyecto tendrá una aplicación basada en sistemas multiagente, en la que los agentes serán los actores del dominio de las emergencias médicas. Estos actores podrán comunicarse entre ellos y, dependiendo de qué tipo sean, podrán realizar cierto tipo de acciones. Estos actores deberán ser racionales, de manera que en base a lo que perciben del entorno y su estado interno decidirán qué hacer.

Además podrá haber organizaciones compuestas por varios miembros, pudiendo así realizar más de una tarea al mismo tiempo.

Se tendrá un simulador capaz de simular el comportamiento del entorno ante acciones de los actores. Dicho entorno será síncrono y con *timesteps* definidos, de manera que por cada *timestep* se enviarán las percepciones a los actores de la aplicación para que estos realicen sus acciones o comunicaciones. Las percepciones se generarán mediante simulaciones, entre las que se encuentran la simulación de movimiento, de enfermedad en los pacientes, y de ocupación en los hospitales. El simulador debe permitir configurar todos los aspectos relativos a la simulación, es decir, la cantidad de cada tipo de actor que habrá en la aplicación incluyendo las características de cada uno de ellos y la duración de la simulación. Por último, toda la información relativa al entorno y a la simulación podrá ser almacenada para poder reutilizarla.

Se creará un visualizador capaz de visualizar una simulación en tiempo real o mediante la interpretación de los datos de una simulación ya realizada, mostrando de una manera visual las interacciones entre los agentes y permitiendo el acceso a la información individual de cada uno de ellos.

3.2.4. Funcionalidad de la aplicación

Según los objetivos dichos al principio y los requisitos especificados anteriormente nuestra aplicación va a tener la siguiente funcionalidad:

- Crear configuración: para poder realizar una simulación de un entorno de emergencias médicas, se deben especificar una serie de parámetros necesarios para dicha simulación. Estos parámetros se definirán en un subsistema del proyecto encargado de solicitar al usuario todos los datos de una nueva configuración de simulación, denominado *mHealth Setup*. Cuando el usuario complete toda esa información requerida, se almacenará de forma

persistente en un fichero denominado "fichero de configuración".

- Simular una configuración: un usuario puede ejecutar una simulación seleccionando un fichero de configuración previamente creado. Con la información que contenga dicho fichero, se comenzará a simular un entorno de emergencias médicas y todo lo que ocurra en esa simulación, se almacenará de forma persistente en un fichero denominado "fichero histórico de simulación".

- Visualizar simulación: mientras se realiza una simulación, el usuario puede elegir ver de forma gráfica lo que está sucediendo en el entorno médico simulado mediante un subsistema de visualización, denominado *Visualizador*. También podrá hacer uso de este módulo sin necesidad de ejecutar una simulación en ese momento, sino que puede cargar en el sistema un fichero histórico previamente creado y visualizar la información generada en la simulación que contenga.

3.3. Arquitectura

Dada la complejidad del proyecto, la arquitectura nos aportará una visión abstracta de alto nivel en la cual podremos diferenciar de manera precisa la estructura del sistema. Esta estructura determina los subsistemas, los componentes y las relaciones que hay entre ellos.

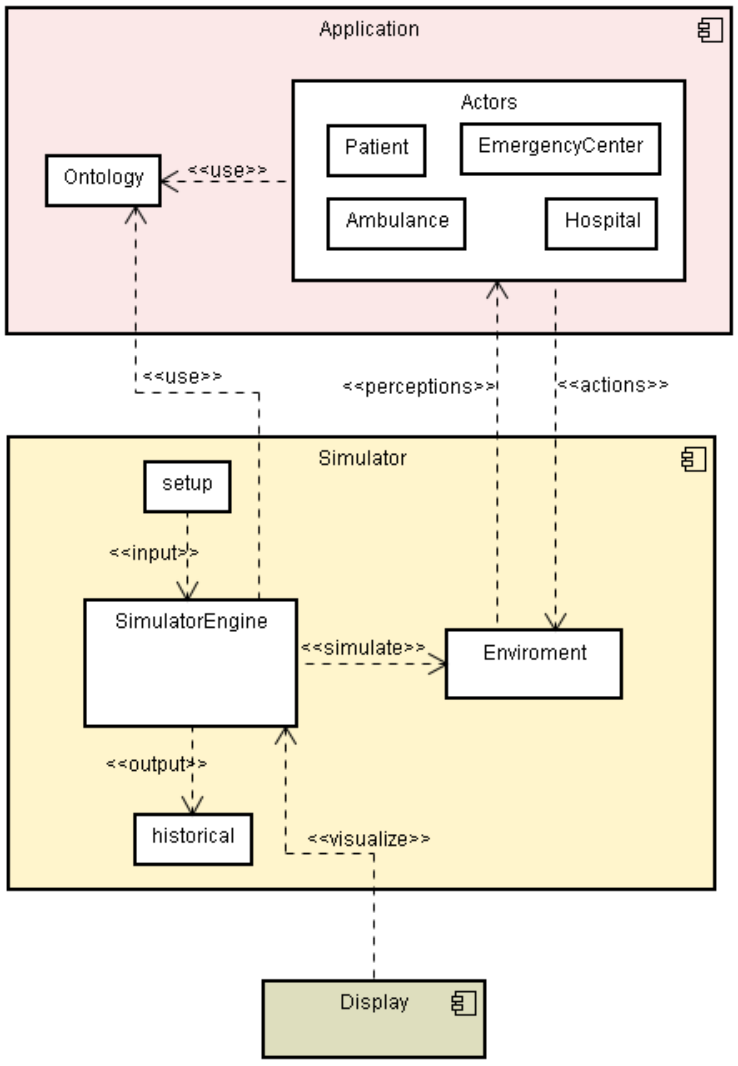


Figura 3.1: Arquitectura

En la figura 3.1 podemos observar los tres subsistemas que conforman la arquitectura de nuestro proyecto. La decisión de tener esta división viene dada, en gran parte, por los requisitos del proyecto, ya que cada subsistema tiene funcionalidades claramente diferenciadas. Además esta división en subsistemas permite implementar de manera independiente cada uno de ellos, lo que significa que podremos tener distintas implementaciones que podrían ser combinadas.

En el caso del subsistema *Aplicación*, la idea es representar el modelo de emergencias del SUMMA de la Comunidad de Madrid, pero este modelo no es el único, existen otros modos de realizar el proceso en los que no hay un centro coordinador que lo gestiona, si no que los propios actores toman sus decisiones en función de otros factores distintos de los actuales, tales como incentivos, etc.

El subsistema *Simulador* es el encargado de simular el entorno en el que se representará el modelo de emergencias de la *Aplicación*, para ello deberá interactuar con dicho subsistema y realizar las simulaciones necesarias para aportar realismo a los sucesos que ocurran en la simulación. Además almacenará los resultados de todas las simulaciones que se realicen para poder analizarlas posteriormente.

El subsistema *Visualizador* se encargará de obtener la información relativa a una simulación almacenada en el subsistema *Simulador* para poder mostrarla de una manera gráfica al usuario. La manera de mostrar esta información dependerá de la tecnología que se utilice en la fase de implementación, pudiendo crear diversos formatos de *Visualizador*.

3.3.1. Aplicación

Contiene todo lo relacionado con los actores involucrados en la simulación, es decir, en este subsistema se encontrarán los pacientes, los centros coordinadores, las ambulancias y los hospitales. Además, contiene la ontología, la cual

es un lenguaje de dominio común que usarán la *Aplicación* y el *Simulador* que servirá para compartir conocimiento y así facilitar la comunicación entre ambos.

- Ontología: en este módulo hay dos conceptos claramente diferenciados: uno son las percepciones y otro son las acciones. Para las percepciones se tiene que definir qué tipo de información es relevante para cada tipo de actor; para las acciones hay que determinar que tipo de acciones puede realizar cada actor. Este módulo será utilizado tanto por la *Aplicación* como por el subsistema *Simulador*, ya que tienen que interactuar entre sí.

- Actores: en este módulo estarán todos los actores de la simulación de emergencias médicas, estos son los pacientes, las centros coordinadores, las ambulancias y los hospitales. La cantidad de actores dependerá de la configuración de la simulación y estos utilizarán la ontología para compartir conocimiento en sus comunicaciones.

3.3.2. Simulador

El objetivo de este subsistema es simular un entorno que interactúe con la *Aplicación* utilizando la misma ontología. Además para la creación de este entorno se necesitarán una serie de parámetros, los cuales se obtendrán del modulo *Setup* de este mismo subsistema. El *Simulador* se compone de diferentes módulos los cuales pasamos a detallar:

- Configuración: proporciona una entrada de datos al motor de simulación, los datos que se obtienen en este módulo son los referentes a la *Aplicación* (pacientes, centros de emergencias, ambulancias, hospitales) y además otros parámetros referentes a la simulación.

- Motor de simulación: se encarga de generar un entorno y realizar las simulaciones necesarias para que el entorno se comporte como un entorno físico

simulado.

- Entorno: es el mecanismo de comunicación entre el *Simulador* y la *Aplicación*. Se encarga de recibir las acciones de los actores de la *Aplicación* y enviarles las percepciones correspondientes una vez simuladas.

- Histórico: almacena todos los datos referentes a una simulación ejecutada para poder utilizarlo y analizarlo posteriormente.

3.3.3. Visualizador

La finalidad de este subsistema es mostrar la información relativa a todos los actores de la *Aplicación* dentro del entorno. Para ello, el *Visualizador* hará uso de la información que genera el subsistema *Simulador*, pudiendo de esta manera, mostrar una visualización de estos datos en tiempo de ejecución "on-line" o mostrar una visualización de una simulación previa "off-line".

Una de las características importantes del *Visualizador* es la capacidad de mostrar al usuario distintos niveles de detalle de la información a visualizar. A nivel de detalle más alto podremos ver el comportamiento del conjunto de actores que interactúan con el entorno, mientras que a un nivel de detalle más bajo podremos ver las características internas de cada actor para poder analizar las decisiones que toman en cada instante.

3.4. Análisis

El objetivo de la fase de análisis es realizar una especificación más precisa de los requisitos, aumentando el nivel de formalismo. Para ello, se utilizarán una serie de diagramas que nos ofrece el lenguaje de modelado *UML*. Con el análisis, se facilita la comprensión, preparación, modificación y mantenimiento de los requerimientos, creando una primera aproximación al modelo de diseño.

3.4.1. Diagramas de casos de uso

Un caso de uso es una descripción de un conjunto de secuencias de acciones que ejecuta un sistema, produciendo un resultado de interés para un actor. Una vez capturados los requisitos, en esta sección identificamos los casos de uso del sistema junto los actores que participan en él. La figura 3.2 muestra el diagrama

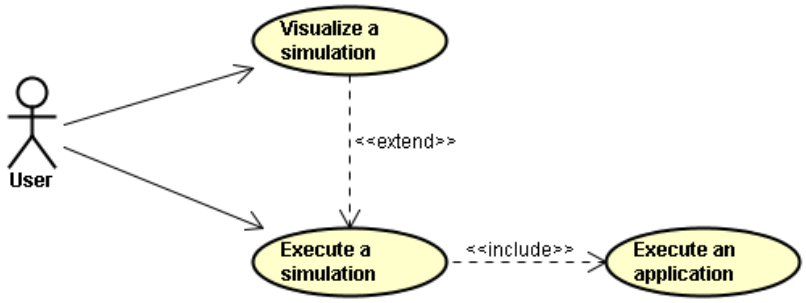


Figura 3.2: Diagrama de casos de usos general

de casos de uso de la aplicación que hemos desarrollado de forma general. Las tres funcionalidades básicas reflejadas en los tres casos de uso, se corresponden con cada uno de los módulos especificados en el apartado de Arquitectura 3.1. El usuario podrá interactuar con los subsistemas *Visualizador* y *Simulador* de manera que pueda realizar una simulación y visualizarla, o directamente ver una simulación generada anteriormente. Con el único subsistema que el usuario no

actúa directamente es con la *Aplicación*, ya que la ejecución de este módulo depende de la ejecución de la simulación. A continuación, se explica de forma más detallada cada caso de uso y su interacción con el usuario:

3.4.1.1. Caso de uso: Ejecutar una aplicación

Este caso de uso comenzará cuando el usuario ejecute una simulación. El usuario no interactúa directamente con este caso de uso, ya que es el módulo *Simulador* es la que inicializa la *Aplicación*, creando cada uno de los componentes necesarios para la ejecución de la simulación. De forma más detallada, en el diagrama 3.3, vemos como en la aplicación se van a tener cuatro tipos de actores: ambulancias, centros de emergencias, hospitales y pacientes. Cada uno de ellos podrá realizar diferentes tareas, algunas de ellas comunes a los cuatro actores, como son "Comunicarse con otros actores" y "Percibir el entorno".

- Caso de uso "Comunicarse con otros actores": engloba todos los posibles envíos de mensajes entre los agentes, como puede ser llamar a un centro de emergencias, solicitar asistencia, preguntar a un hospital por su disponibilidad, etc. Estos mensajes serán específicos de cada actor, y serán detallados más adelante (ver apartado 3.4.3).

- Caso de uso "Percibir entorno": será específica para cada actor, ya que cada uno recibirá únicamente la información que necesite para posteriormente, decidir que tarea hacer.

Las demás funcionalidades son:

- Caso de uso "Moverse": tanto pacientes como ambulancias pueden moverse por el entorno, este movimiento equivale a un movimiento físico en el entorno, teniendo en cuenta velocidades, distancias, tiempos, y zonas por las que se puede circular.

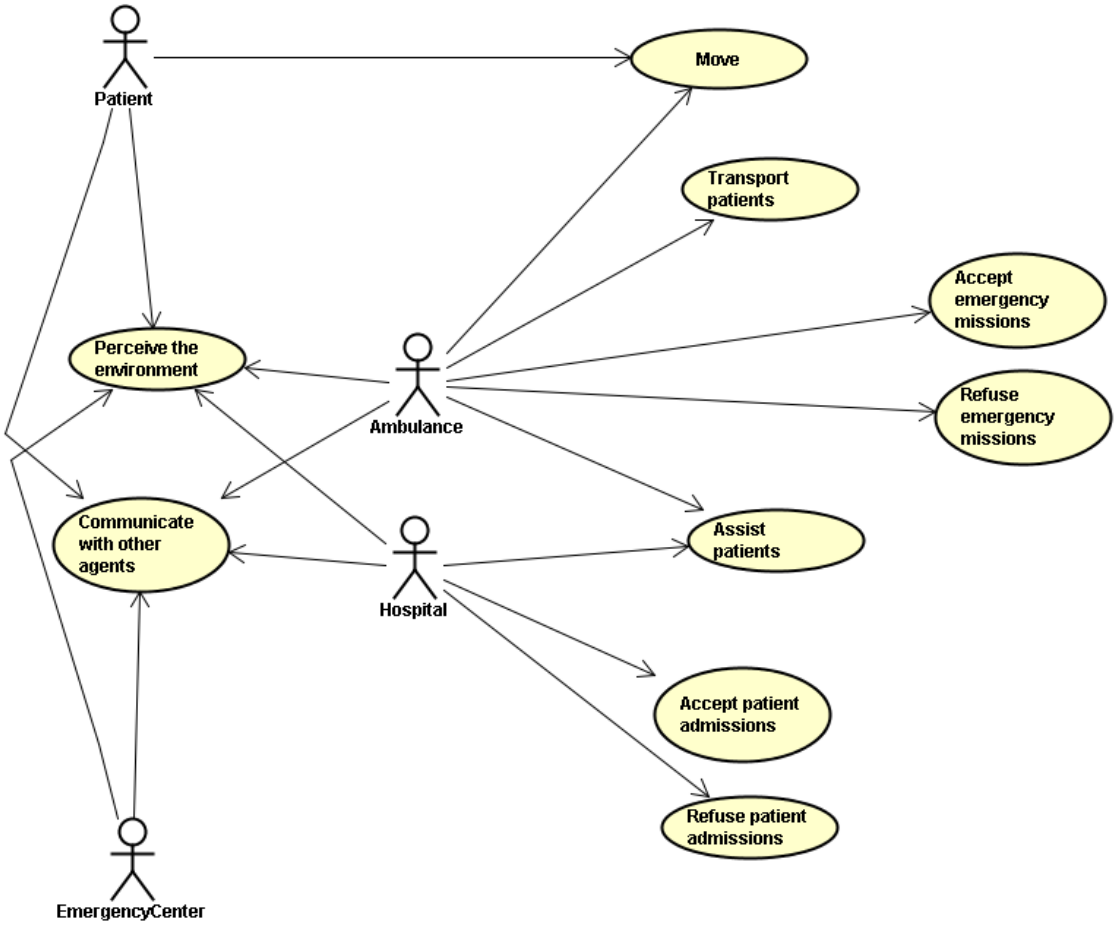


Figura 3.3: Casos de uso de la aplicación

- Caso de uso "Transportar pacientes": este caso de uso sólo lo puede realizar una ambulancia. Ésta es capaz de llevar un paciente desde un origen a un destino, concretamente en nuestro sistema, desde la localización del paciente hasta un hospital.

- Caso de uso "Aceptar misión de emergencia médica": los centros coordinadores asignarán las misiones de emergencia médica a las diferentes ambulancias que existan en el entorno. Estas podrán aceptar la misión y realizar posteriormente todas las tareas necesarias para conseguir curar al paciente

asignado.

- Caso de uso "Rechazar misión de emergencia médica": las ambulancias tendrán sus propios criterios para aceptar una misión de emergencia médica o no. Por ello, una ambulancia también puede rechazar la misión que el centro coordinador quiera asignarle, pudiendo deberse a que la ambulancia ya está asignada, no tiene un equipo médico adecuado u otros factores.

- Caso de uso "Asistir pacientes": Tanto ambulancias como hospitales se componen de equipos médicos que son capaces de asistir pacientes con el fin de curarles.

- Caso de uso "Aceptar la admisión de pacientes": cuando un paciente enferma, puede requerir de atención hospitalaria. Los hospitales, según sus propios criterios, podrán admitir a los pacientes para realizar una asistencia médica.

- Caso de uso "Denegar la admisión de pacientes": al igual que un hospital puede admitir pacientes, también puede rechazar la admisión de éstos, ya sea por total ocupación de sus camas, la escasez de recursos para realizar la asistencia u otros factores.

3.4.1.2. Caso de uso: Ejecutar una simulación

El caso de uso principal con el que interactúa el usuario es "Ejecutar una simulación". Este caso de uso se encarga de realizar la simulación de un escenario de emergencias médicas. Para ello, necesita hacer uso del módulo *Aplicación*, ya que en el entorno de la simulación van a participar los actores definidos en el diagrama 3.3. Con mayor nivel de detalle, se ve en el diagrama de casos de uso 3.4 las funcionalidades en las que se descompone realizar una simulación. En este

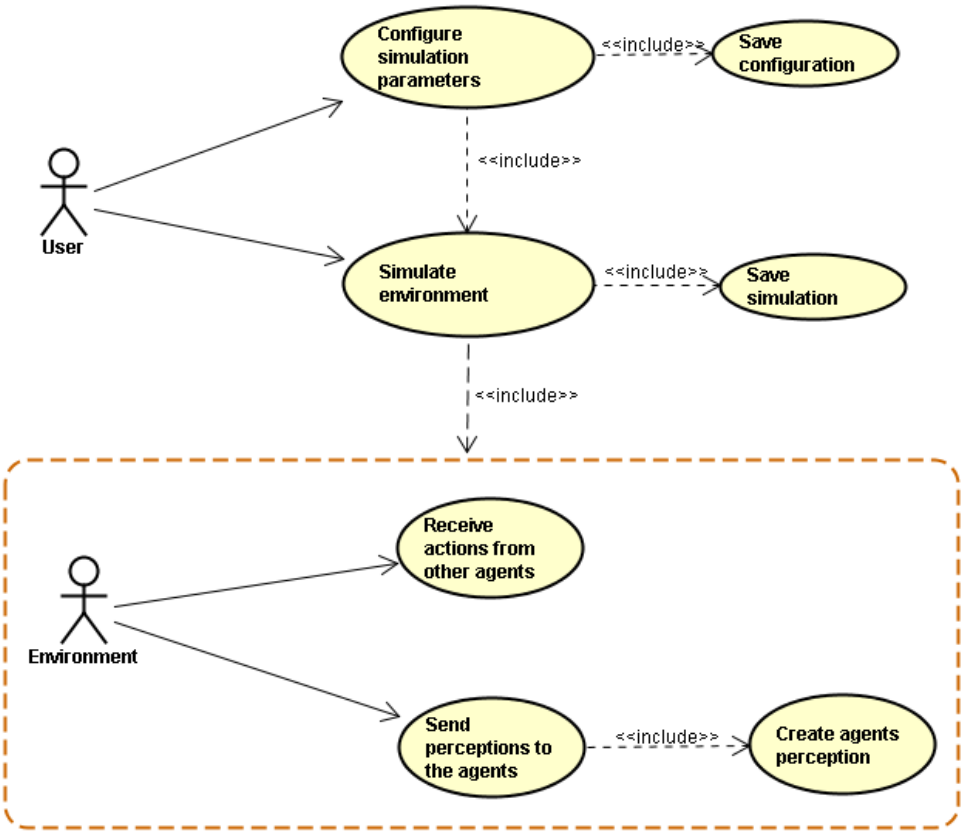


Figura 3.4: Casos de uso del simulador

caso de uso se van a tener dos tipos de actores: el usuario de la aplicación y un "Entorno" de emergencias médicas. La descripción de cada caso de uso se muestra a continuación:

- Caso de uso "Configurar los parámetros de una simulación": el usuario podrá seleccionar si quiere configurar los diferentes parámetros de un escenario de emergencias médicas. Este caso de uso se podrá realizar antes de la ejecución de una simulación, de forma que al completar una configuración, se comience automáticamente el caso de uso "Simular entorno" con los parámetros introducidos por el usuario en esa configuración.

- Caso de uso "Salvar configuración": toda la configuración que el usuario genere en el caso de uso "Configurar los parámetros de una simulación", se almacenará en un fichero de configuración automáticamente al completar el proceso anterior.

- Caso de uso "Simular entorno": el usuario podrá elegir si desea ejecutar una simulación directamente sin pasar por el caso de uso "Configurar los parámetros de una simulación". En este caso, necesita cargar un fichero de configuración válido que se haya creado anteriormente con el programa y se procederá a ejecutar la simulación asociada al fichero. Este caso de uso también tendrá lugar cuando el usuario quiere configurar una nueva simulación, ya que en ese caso de uso se incluye a éste.

  Dentro de la simulación, nos encontramos con otro actor, el "Entorno", que tendrá tres funcionalidades básicas. Estos tres casos de uso forman parte de la ejecución de la simulación:

  - Caso de uso "Recibir acciones": el Entorno será el encargado de recibir las acciones que deseen ejecutar los actores del módulo *Aplicación*. Como estos actores se comunican unos con otros a través del entorno definido, toda acción y comunicación entre agentes debe pasar por este módulo *Simulador*, ya que es quien contiene al actor "Entorno".

  - Caso de uso "Crear percepciones": una vez simuladas las acciones que se reciben en el *Simulador*, el Entorno debe generar las percepciones adecuadas de cada actor según se haya modificado el estado del mundo tras la simulación de las acciones recibidas.

  - Caso de uso "Enviar percepciones": las percepciones creadas por el Entorno se enviarán a cada actor del subsistema *Aplicación*, informándoles del nuevo estado del mundo tras la simulación de sus acciones, para

que estos puedan tomar nuevas decisiones y solicitar acciones a emular.

- Caso de uso "Salvar simulación": mientras se realiza una simulación, siempre se irá guardando la información generada en un fichero histórico. Este tipo de ficheros se podrán luego visualizar con el módulo *Visualizador*.

3.4.1.3. Caso de uso: Visualizar una simulación

Este caso de uso es el encargado de mostrar al usuario todo el proceso de simulación de forma gráfica. Al realizar una simulación, el usuario tendrá la opción de elegir si quiere que se le muestre la interfaz gráfica mientras se simula o no. En el caso de que quiera verla, este caso de uso será una extensión de la ejecución de la simulación. Si no desea verla en ese momento, este caso de uso no tendrá lugar, pudiendo verse la simulación desarrollada posteriormente mediante la carga del fichero de su histórico. Visualizar una simulación se ve con más detalle en el

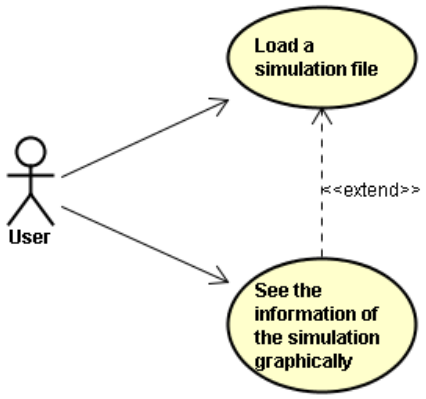


Figura 3.5: Casos de uso del visualizador

diagrama 3.5. Las funcionalidades que recoge este subsistema son:

- Caso de uso "Ver la simulación gráficamente": el usuario podrá ver lo que va pasando en el entorno y en cada actor que compone la *Aplicación* en una interfáz gráfica.

- Caso de uso "Cargar un fichero de simulación": el usuario podrá cargar una simulación ya realizada, almacenada por el módulo *Simulador*, y visualizarla de manera "off-line", ya que el caso de uso "cargar simulación" incluye a "visualizar simulación".

3.4.2. Simulador y visualizador

Mediante la utilización de diagramas de estados, se definen comportamientos importantes del sistema de forma más minuciosa, detallando las secuencias de estados por las que pasan casos de uso u objetos a lo largo de su vida, junto con las acciones o eventos que hacen que se produzcan esas transiciones de estados.

En esta sección se describe la relación entre los casos de uso *Visualizar una simulación* y *Ejecutar una simulación* descritos en la sección anterior. Aquí se observa como actuaría el sistema en el caso de que el usuario decida ver una simulación de forma gráfica en tiempo real, o prefiera ejecutarla, guardarla y verla posteriormente. En el primer caso, la funcionalidad *Visualizar una simulación* sería una extensión de *Ejecutar una simulación* y en el segundo, el caso de uso *Visualizar una simulación* sería independiente del subsistema *Simulador*.

En el diagrama de estados 3.6 se describe la secuencia de estados, acciones y eventos que suceden cuando el usuario elige la opción de realizar una simulación:

El subsistema *Simulador* creará el estado inicial del entorno a partir de los datos leídos del fichero de configuración elegido e inmediatamente empezará el proceso de simulación. Se guardará en un fichero histórico el primer paso, el cual sólo se compondrá del estado del entorno inicial. A partir de ese estado, se creará el propio entorno y los actores que participarán en él.

En ese momento, se comprobará si el usuario eligió la opción de visualización junto con la simulación o no. En el caso de que la haya elegido, se inicializará la interfaz gráfica del subsistema *Visualizador* y se mostrará en ella el primer paso de

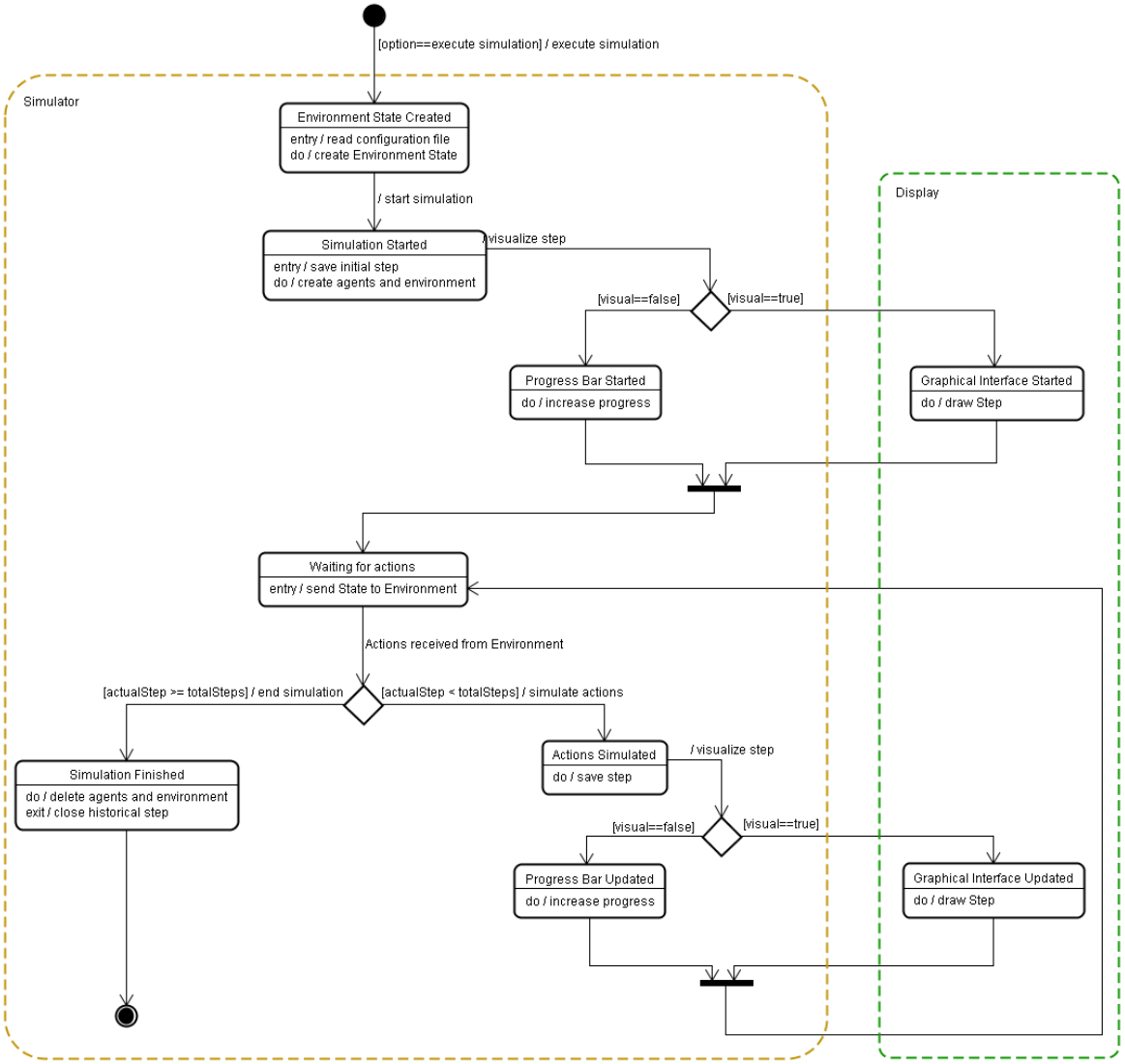


Figura 3.6: Diagrama de estados de la simulación *online*

la simulación de forma adecuada. Si su opción fue no visualizarla, se creará una barra de progreso informativa y se incrementará su desplazamiento según sea necesario.

Después de que se realice alguna de esas dos acciones, el simulador pasará a un estado de espera hasta recibir una lista de acciones por parte del entorno. Justo al entrar a este estado, se enviará al entorno su estado interno, para que él

sepa actualizar las percepciones de los demás actores y ellos le respondan con las acciones que quieren realizar (ver los casos de uso del entorno en el diagrama 3.4). El entorno esperará el tiempo indicado por el usuario para completar una lista de acciones y, pasado ese tiempo, se las mandará al simulador para que éste las procese. Para ello, el simulador comprobará si el número de iteraciones "enviar percepciones - recibir acciones" es superior al número total de pasos que va a tener la simulación. Si es igual o superior, se finalizará la ejecución eliminando todos los actores del entorno y al propio entorno para que se corte la comunicación entre ellos y se cerrará el fichero con el histórico de pasos, guardado en su ubicación correspondiente. Si el número de iteraciones es inferior, se simulan las acciones y se guarda toda la información relevante del nuevo paso simulado.

De nuevo, se comprobará si se quiere una simulación visual o no, y se dibujará en la interfaz o se aumentará la barra de progreso según corresponda. Después, el simulador volverá a un estado de espera, mandando al entorno su estado actualizado tras la simulación de las acciones. Este ciclo se repetirá hasta que se realicen tantos pasos de simulación como el usuario indicó en la configuración inicial.

En el diagrama de estados 3.7 se muestra el comportamiento del sistema cuando el usuario elige la opción de visualizar una simulación previamente creada:

La primera acción que se hace es leer el fichero seleccionado por el usuario, el cual contendrá el histórico de la simulación que se quiere visualizar. De él, se leerán un conjunto de datos y se comprobará si esos datos se corresponden con un paso de simulación válido o no.

Si no es válido, se informará al usuario de el error y se finalizará el proceso. Si es válido, se inicializará la interfaz gráfica y se mostrarán los datos que se pueden representar en ella.

Después, el visualizador pasará a un estado de espera durante el tiempo

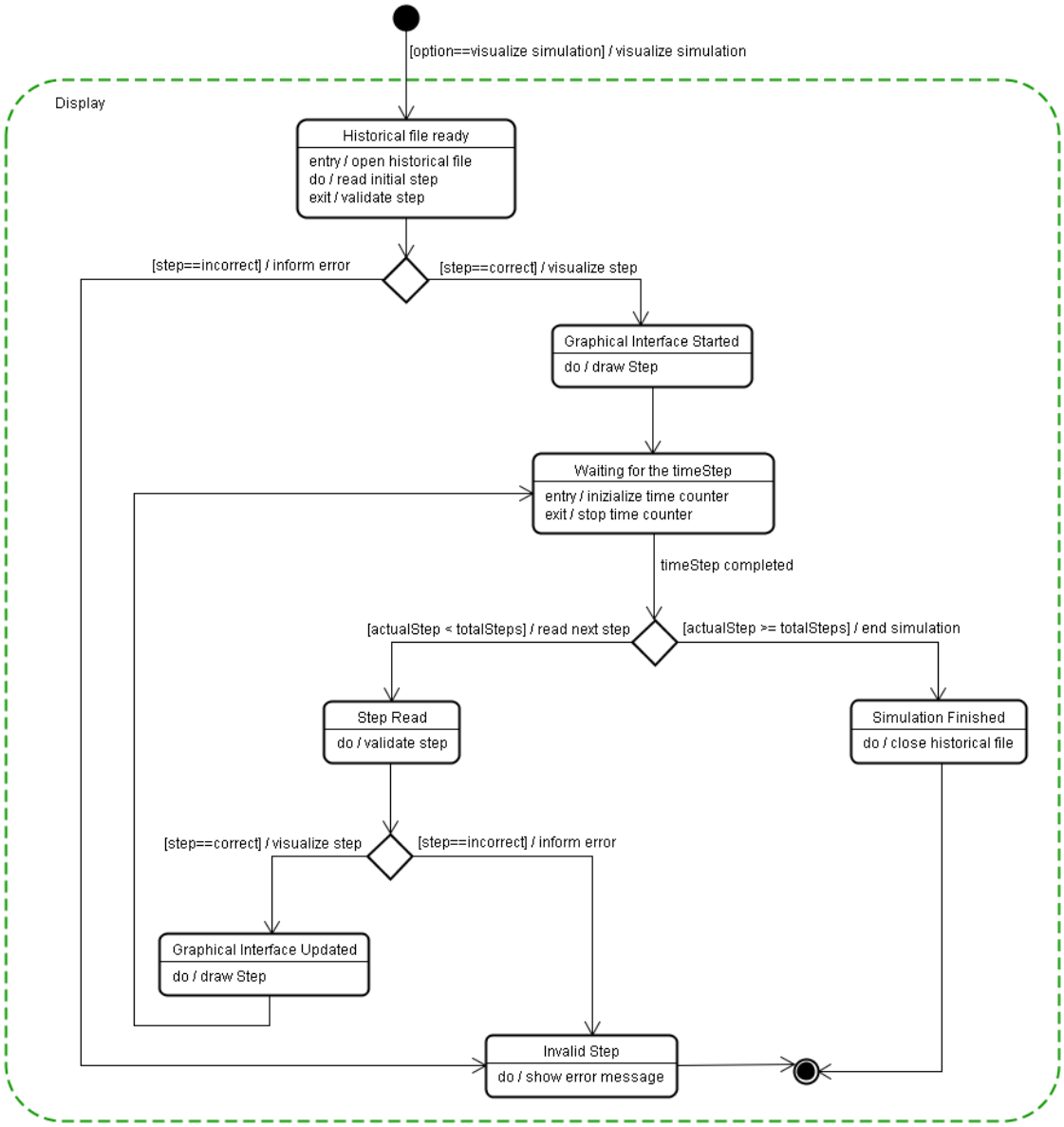


Figura 3.7: Diagrama de estados de la simulación *offline*

definido en el fichero histórico como *timestep*. Para ello, se activará un contador de tiempo y cuando éste marque el tiempo del *timestep*, el sistema comprobará si estamos en un número de paso de simulación correcto o si éste supera al número total de pasos de la simulación indicado en el fichero leído. Si se supera el número

de pasos, el fichero histórico no tendrá más información para leer y se finalizará la visualización de la simulación cargada. De lo contrario, se leerán otro conjunto de datos y se procederá a su validación.

Si el paso es correcto, se llama otra vez a la interfaz gráfica para que muestre la información y se repite el ciclo anteriormente explicado tantas veces como pasos de simulación existan. Si algún paso es incorrecto, se muestra un mensaje de error y se finaliza la visualización.

3.4.3. Aplicación con simulador (Agentes)

En esta sección se explicará como interactúa el subsistema aplicación con el subsistema simulador. Esta interacción se basa en el envío de acciones o comunicaciones por parte de los actores de la aplicación al entorno del simulador y este a su vez enviará las percepciones una vez simuladas las acciones.

Como podemos observar en el diagrama 3.8, el envío de percepciones y acciones se realiza tantas veces como este definido el *timestep* de la simulación. Cuando un actor recibe su percepción debe actualizar su estado interno, de esta manera será capaz de tomar la decisión sobre que acción realizar de una manera más efectiva.

El protocolo utilizado para la comunicación entre el *Simulador* y la *Aplicación* es una variación del protocolo *request* (1.4.2.1) detallado en el estándar FIPA. Los actores de la *Aplicación* solicitan al entorno que realice una acción, en caso de utilizar el protocolo *request* el entorno les respondería indicando la aceptación (*agree*) o no (*refuse*) de la petición y en caso afirmativo enviaría otro mensaje informando si la acción se ha realizado con éxito (*inform-done* o *inform-result*) o no (*failure*). De esta manera los actores de la *Aplicación* sabrían en cada momento el resultado de sus acciones sobre el entorno, pero este no es el comportamiento que deseamos. Los propios actores deben determinar el resultado de sus acciones

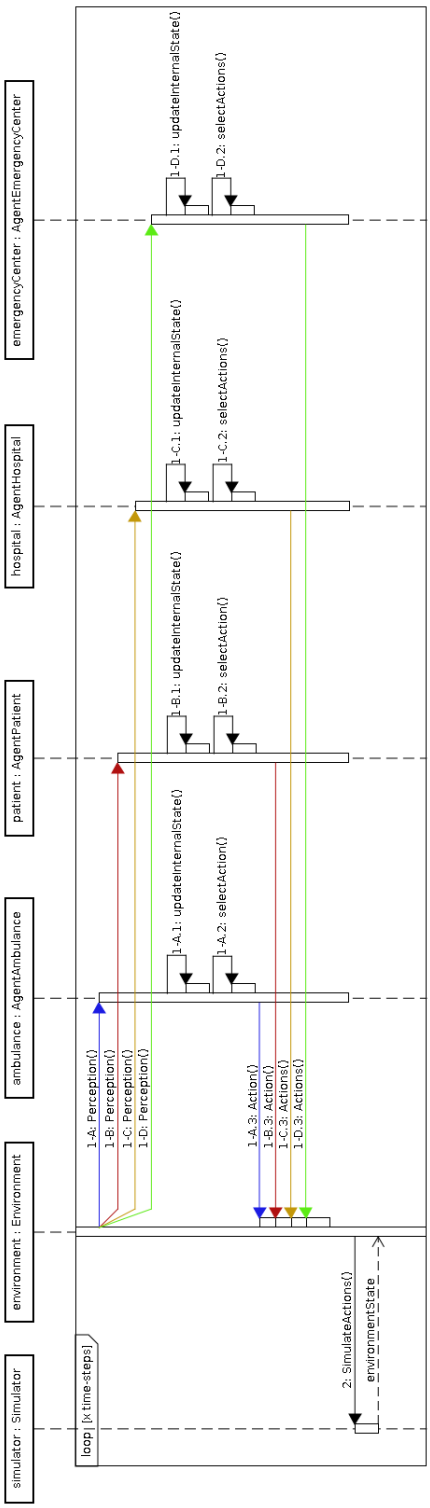


Figura 3.8: Interacción entre aplicación y simulador

mediante sus percepciones, por ello el entorno, una vez recibida la solicitud *request* y obtenidos los resultados de la simulación de las acciones, informara a los actores mediante un *inform* con sus nuevas percepciones.

A continuación se analizará de manera más detalla las diversas decisiones que deben tomar los actores.

3.4.3.1. Paciente

Para comprender las decisiones de un paciente hay que tener en cuenta que su único objetivo es curarse, por ello, una vez que percibe que esta enfermo tiene que tomar la decisión de con que centro de emergencias contactar para solicitar asistencia medica. Una vez realizada la comunicación, el paciente deberá esperar hasta recibir la contestación del centro de emergencias, el cual le comunicará que debe esperar a que la ambulancia llegue hasta donde se encuentre. A partir de este momento el paciente esperará hasta ser curado y no tendrá que tomar más decisiones. Este flujo de interacciones lo podemos ver en la figura 3.9.

3.4.3.2. Centro de emergencias

El centro de emergencias es el encargado de mediar entre pacientes y ambulancias, para ello recibirá llamadas de pacientes y localizará ambulancias para que realicen asistencias medicas.

En la figura 3.10 podemos observar que decisiones debe tomar el centro de emergencias una vez que recibe la llamada de un paciente. Cuando se recibe esta llamada el centro añadirá a su sistema la solicitud de ese paciente hasta que se le asigne una ambulancia y se le avisará de que se mantenga a la espera. Seguidamente se solicitará a una ambulancia que realice una asistencia al paciente, la elección de la ambulancia se basa en la eficiencia para tratar la emergencia del paciente y ademas se tendrá en cuenta su cercanía. A partir de este momento

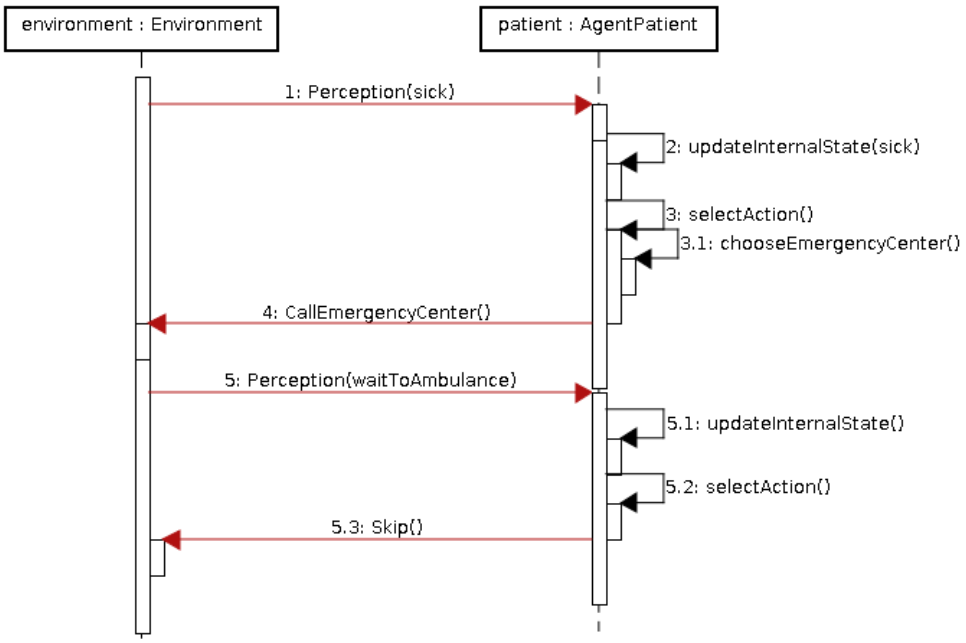


Figura 3.9: Decisiones del paciente

el centro de emergencias seguirá realizando tareas tales como atender nuevas llamadas hasta que se reciba la aceptación de la misión por parte de la ambulancia y añadiendo al sistema la misión que se realizará al paciente.

Respecto a las misiones que se estén realizando los centros coordinadores deberán ser informados del éxito o el fracaso de las asistencias, por ello también serán capaces de tratar informes de finalización de misión enviados por las ambulancias.

3.4.3.3. Ambulancia

La ambulancia, a diferencia de los otros actores, tiene varios objetivos. El primero de ellos es proporcionar asistencia medica a un paciente *in situ*, en caso de no lograr curar al paciente se accede al segundo objetivo el cual trata de trasportar el paciente a un hospital.

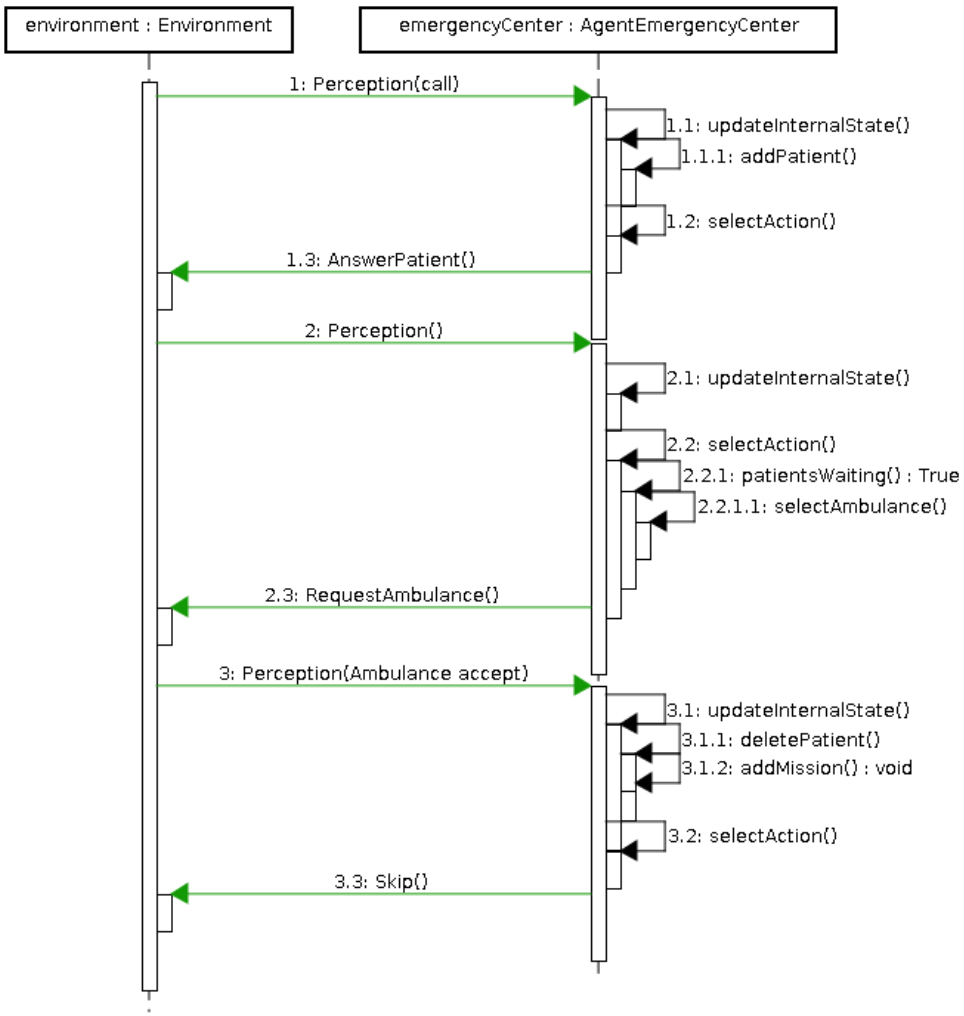


Figura 3.10: Decisiones del centro de emergencias

En la figura 3.11 podemos ver la interacciones de una ambulancia desde que recibe una percepción con una asignación de misión. Una vez que una ambulancia recibe una solicitud para asistir a un paciente del centro de emergencias deberá decidir si aceptar o no la misión, para tomar esta decisión tendrá que ver si esta disponible o no. Una vez aceptada la misión se moverá hacia la localización del paciente y allí realizará la asistencia *in situ*, la duración de esta asistencia es una de las decisiones que deberá tomar la ambulancia en base a la adecuación

del equipo a la emergencia tratada. A partir de aquí pueden darse las siguientes situaciones: la ambulancia ha sido capaz de curar al paciente y deberá finalizar la misión informando al centro de emergencias, la ambulancia no ha sido capaz de curarlo y debe de transportarlo a un hospital. En este ultimo caso la ambulancia debe de tomar la decisión de que hospital seleccionar para que traten al paciente, al igual que los centros de emergencias los factores que tendrá en cuenta para esta decisión serán el nivel de emergencia que se trata y la cercanía del hospital. Una vez decidido el hospital la ambulancia se dirigirá a él y el paciente accederá al mismo, en este instante la ambulancia termina la misión.

Existen algunos caminos alternativos a estas secuencias de acciones ya que los pacientes pueden fallecer. En el caso de que ocurra esta situación se pueden dar dos casos. Mientras que la ambulancia esta dirigiéndose hacia la localización del paciente, este fallece, por lo que cuando la ambulancia llega al lugar percibe el estado de salud del paciente y deberá finalizar la misión. El otro caso es mientras se esta transportando el paciente al hospital, la ambulancia continuará su camino hasta llegar al hospital, el cual se hará cargo del paciente fallecido.

3.4.3.4. Hospital

El objetivo del hospital es curar a los pacientes que tiene ingresados. Para ello, además de los pacientes externos a la aplicación los cuales serán simulados, deberá ser capaz de aceptar nuevos pacientes mediante comunicaciones con ambulancias. La decisión de aceptar o no un paciente se toma teniendo en cuenta la ocupación del hospital.

Para que un paciente sea admitido en un hospital previamente la ambulancia debe comunicarse con él para reservar acceso al mismo. El sentido de esta comunicación previa es evitar que una vez que la ambulancia llegue a un hospital no halla camas disponibles y tenga que dirigirse a cualquier otro. El uso de esta

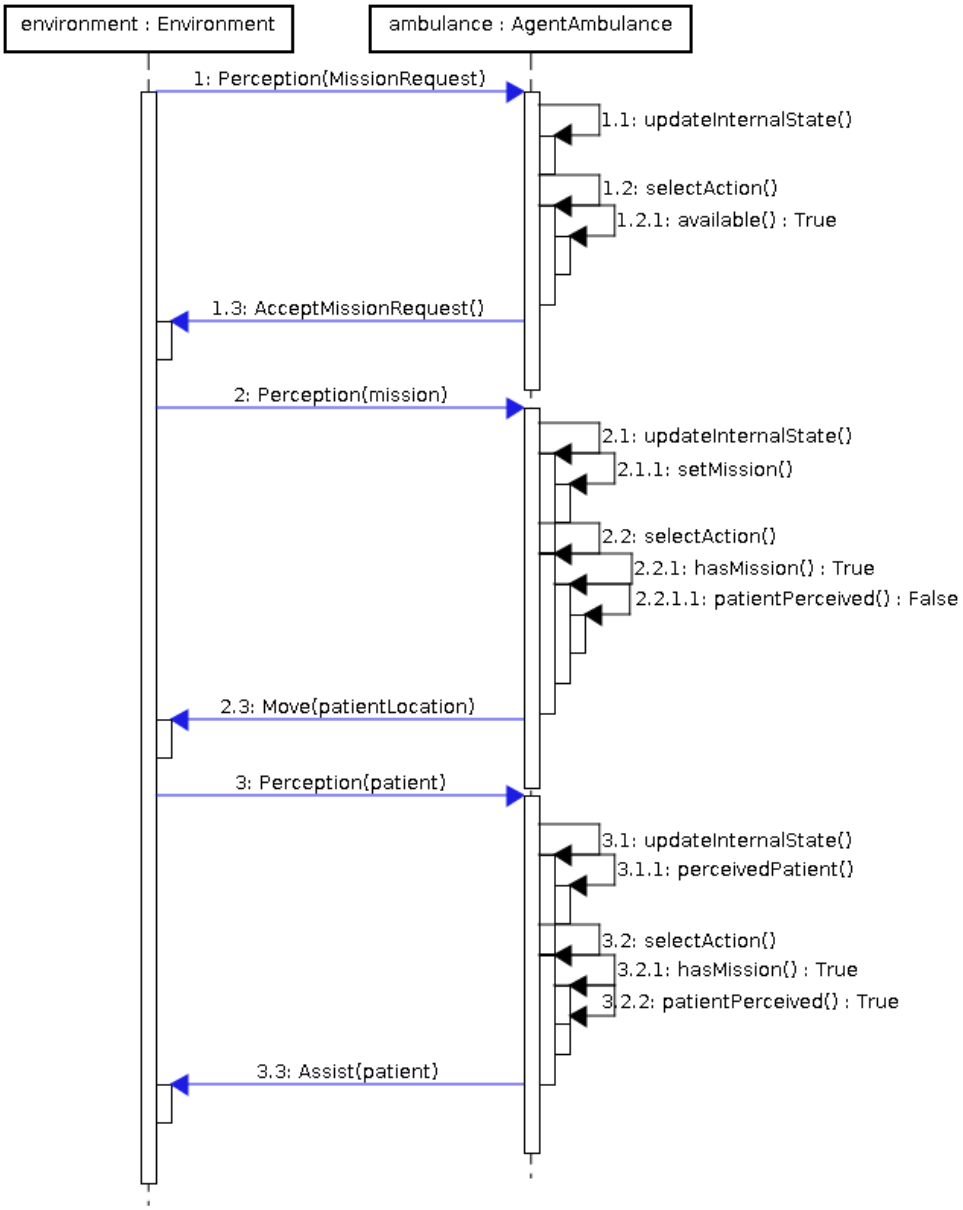


Figura 3.11: Decisiones de la ambulancia

comunicación no implica la admisión del paciente ya que puede darse el caso que aún aceptando la reserva el paciente no pueda ser admitido en el hospital debido al factor de ocupación simulado por parte del simulador, por lo que la ambulancia deberá dirigirse a otro hospital. Una vez se realiza la admisión de un paciente el

hospital tratará de curarlo y así poderle dar el alta médica. El flujo básico para que un paciente sea admitido lo podemos ver en la figura 3.12

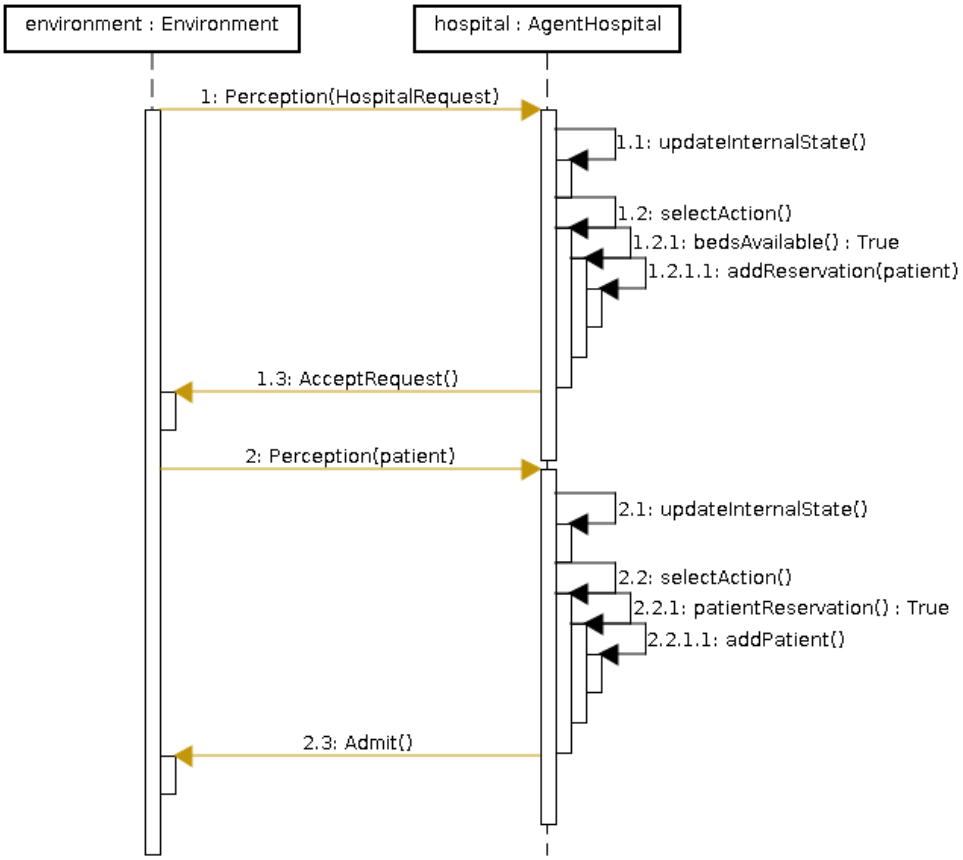


Figura 3.12: Decisiones del hospital

3.5. Diseño

En esta sección dotaremos de un modelo de diseño a nuestro proyecto en el cual apreciaremos la transformación que tiene toda la interacción descrita a nivel de análisis a un modelo concreto. Para ello identificaremos las diferentes clases que existen en cada modulo de cada subsistema y estableceremos las relaciones que hay entre cada una de ellas.

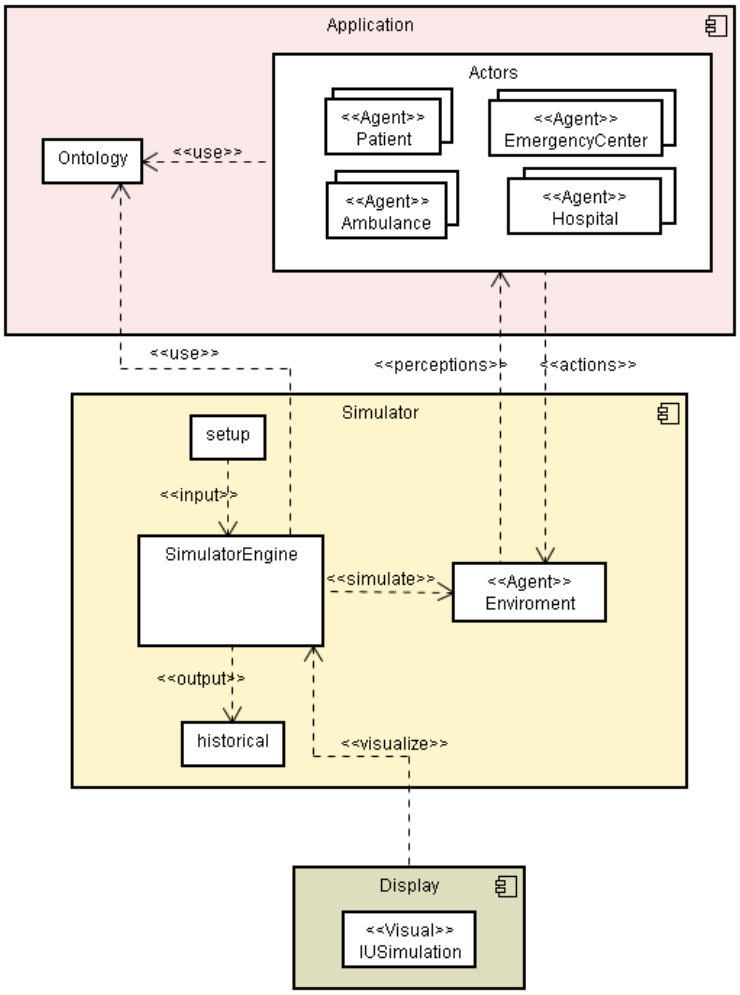


Figura 3.13: Diseño a bajo nivel

Uno de los aspectos importantes a destacar en esta etapa es la utilización de agentes software para modelar ciertos módulos del sistema, ver 3.13. Utilizar agentes como actores del subsistema *Aplicación* permite modelar comportamientos de manera sencilla para realizar acciones sobre el entorno y recibir las percepciones. Por este mismo motivo, el entorno del simulador será modelado como un agente, además, la comunicación entre los dos subsistemas se verá facilitada ya que se utilizarán comunicaciones del mismo tipo.

3.5.1. Diagramas de clases

En esta sección detallaremos las clases del sistema y sus interrelaciones por cada uno de los subsistemas del proyecto.

3.5.1.1. Aplicación

El subsistema *Aplicación* se compone de una ontología y actores del dominio de las emergencias médicas, para el diseño de estos componentes se utilizarán agentes software. A continuación se detallan los elementos más importantes de la aplicación.

3.5.1.1.1. Agentes

Los agentes de la *Aplicación* son agentes JADE, todos ellos utilizan la misma ontología definida en la *Aplicación* y utilizan un codec *FIPA SL* mediante el cual se realizan las conversiones del lenguaje contenido. Como se puede observar en la figura 3.14, los agentes pueden ser de dos tipos, organizaciones o miembros, los cuales detallamos a continuación:

- Organización: las organizaciones con las que nos encontramos en el dominio del proyecto son dos: los centros de emergencias y los hospitales. En los

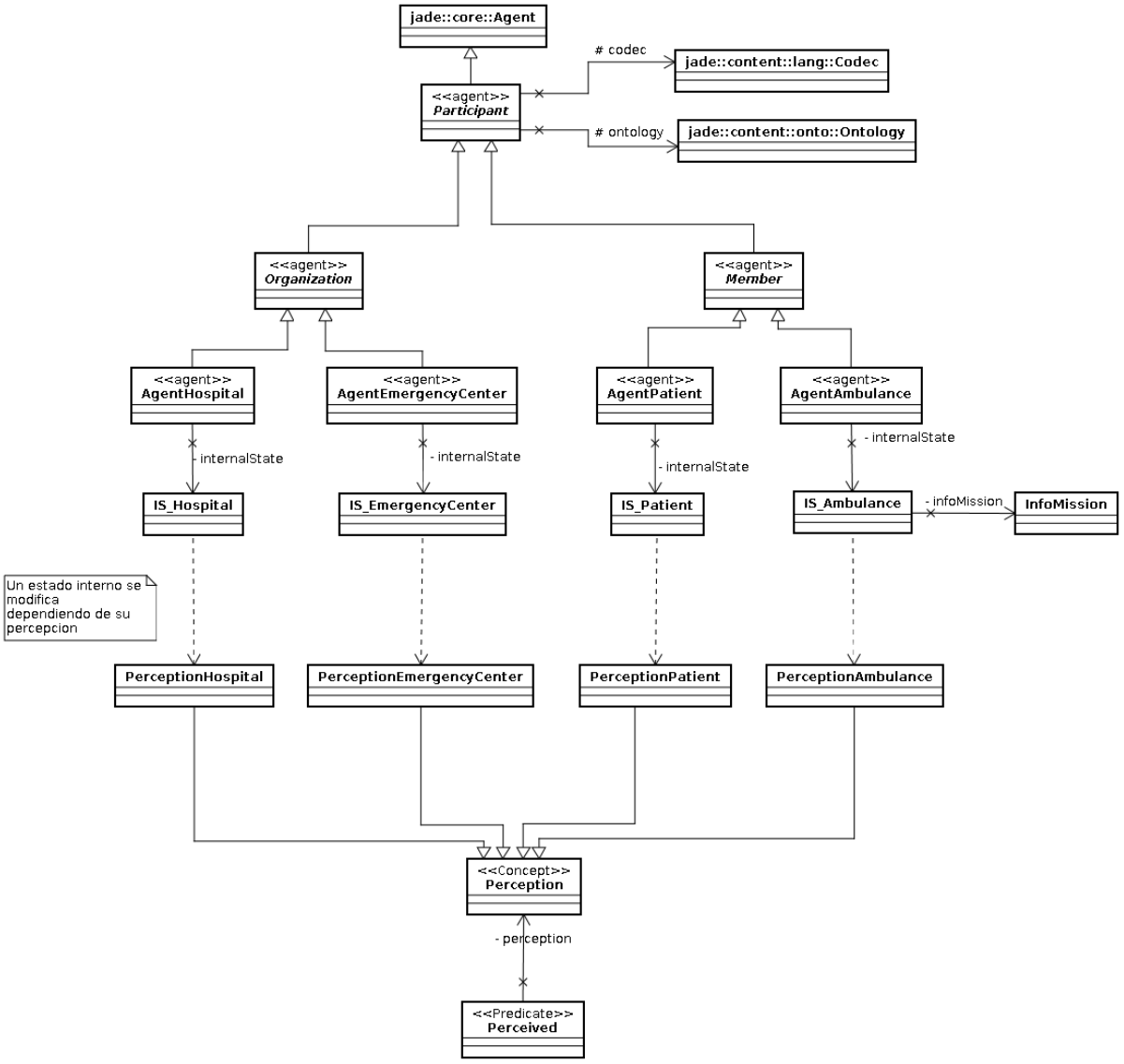


Figura 3.14: Agentes y estados internos

centros de emergencias hay operadores los cuales se encargan de atender llamadas de pacientes y encolarlos en el sistema de espera del centro, por ello la función de los distintos operadores es la misma, por lo que se pueden considerar un solo agente capaz de realizar varias acciones. El mismo caso se da en los hospitales, los cuales deben ser capaces de admitir varios

pacientes a la vez y a su vez manejar comunicaciones, etc... Por todo ello una organización es idéntica a un miembro de la *Aplicación* pero con la capacidad de realizar tantas acciones como miembros internos posea en un mismo *timestep*.

- Miembro: los miembros del sistema son los pacientes y las ambulancias, los cuales solo pueden realizar una acción en cada *timestep*.

Los agentes poseen un comportamiento (*behaviour*) que se compone de cuatro subtareas: recibir percepción, procesar percepción, seleccionar siguiente acción, y enviarla. Tanto recibir percepción, como enviar acción son tareas comunes para todos los agentes, el resto de tareas son dependientes de cada tipo de agente. Cada agente posee un estado interno en el cual almacenará información relativa a sus propias características y a lo que conoce del entorno que le rodea, se podría decir que el estado interno de un agente es su conocimiento. Gracias a este estado interno el agente irá modificando su conocimiento en base a las percepciones y las acciones que realiza, pudiendo determinar cual es la mejor acción a realizar en cada *timestep*.

3.5.1.1.2. Acciones

Los agentes de la *Aplicación* pueden realizar acciones. Estas acciones implementan la interfaz `AgentAction` de *Jade* y no todas pueden ser realizadas por cualquier agente, ver 3.15. Podemos distinguir dos grandes tipos de acciones: unas son las acciones que repercuten de alguna manera en el entorno, a las que denominaremos "acciones", y otras son las que no influyen en el entorno, estas son las "comunicaciones".

- Acciones: la realización de una acción implica un cambio en el entorno (en el presente proyecto, simulado por el motor de simulación). Este cambio

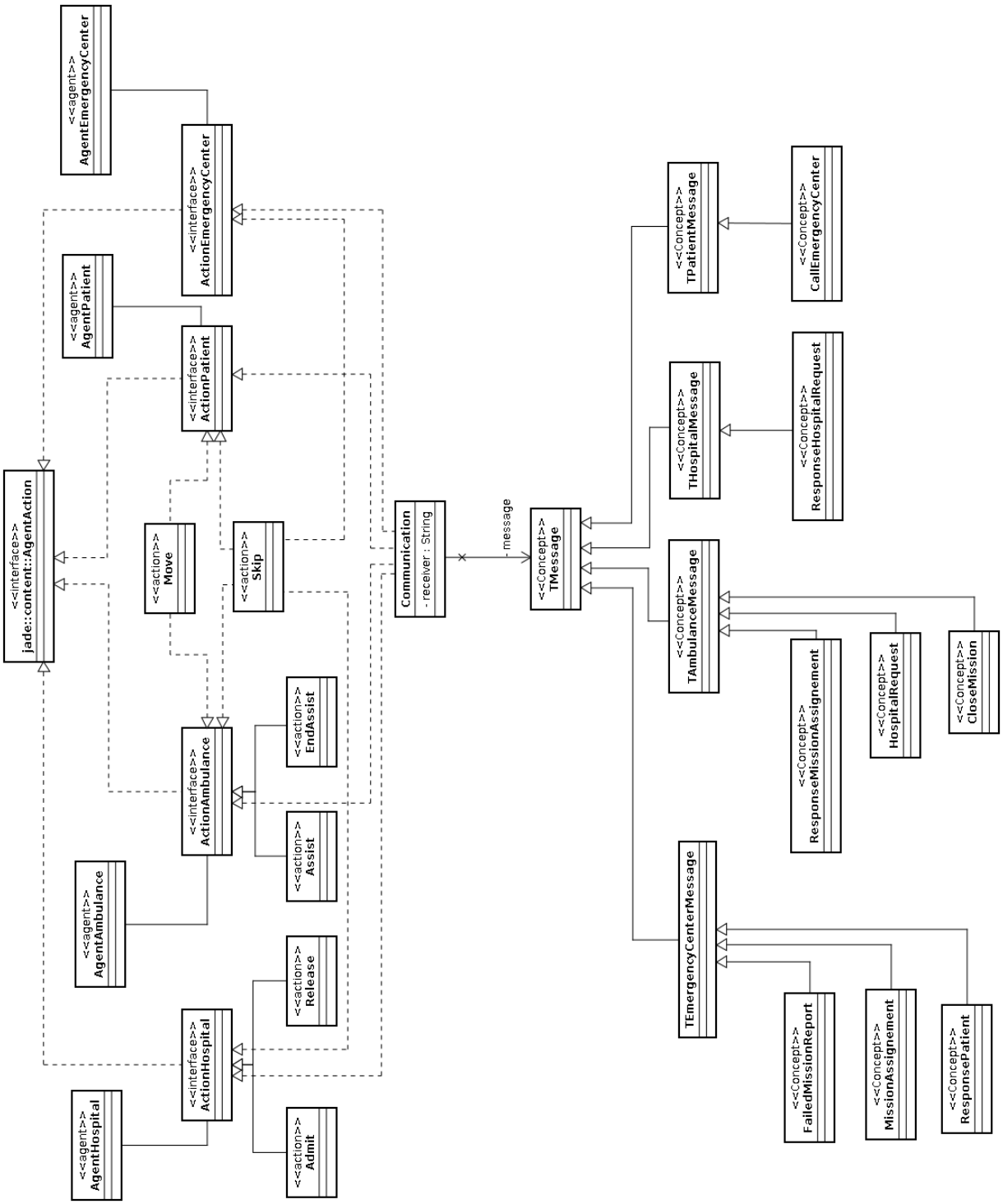


Figura 3.15: Acciones

podrá influir en varios agentes de la *Aplicación*, como por ejemplo, la realización de un *Assist* influye tanto en un agente ambulancia, como en un agente paciente.

- Comunicaciones: el envío de una comunicación es un acto de intercambio de información entre agentes, por lo que el entorno simplemente propaga la información desde un emisor hasta un receptor.

Tanto las acciones como las comunicaciones están descritas en el mismo lenguaje pero para que los agentes se entiendan entre sí, necesitan que el contenido de esa comunicación sea entendible por todos, por ello se utiliza la ontología de las emergencias medicas para que todos ellos compartan el mismo significado de los conceptos que se intercambian.

A continuación se explica la semántica de cada tipo acción:

- `Admit`: la utilizan los hospitales para ingresar un paciente en sus instalaciones. Para que esta acción se realice con éxito tienen que darse ciertas condiciones: debe haber camas libres y el paciente que se desea admitir debe encontrase en una ambulancia que ha recibido un `ResponseHospitalRequest` afirmativo y cuya localización corresponde con la del hospital.

- `Release`: la realizan los hospitales para dar de alta a un paciente ingresado.

- `Assist`: es utilizada por las ambulancias cuando quieren realizar una asistencia *in situ*. Cuando una ambulancia ha llegado a la localización en la que se encuentra el paciente que se le ha asignado en su misión podrá realizar esta acción siempre y cuando el paciente este enfermo.

- `EndAssist`: sirve para terminar la asistencia *in situ* a un paciente. La realización de esta acción se permite a ambulancias que previamente `Assist` sobre el mismo paciente.

- `Move`: esta acción la pueden realizar tantos los pacientes como las ambulancias, se utiliza para trasladarse de una localización a otra. El realizar esta acción no implica llegar al destino solicitado, ya que se debe simular el movimiento hacia dicho lugar.

- `Skip`: esta acción la pueden realizar todos los agentes de la aplicación y se utiliza cuando no se desea realizar otro tipo de acción, es equivalente a no hacer nada.

- `Communication`: a continuación se detallar los distintos contenidos que puede tener la comunicación.

    - `FailedMissionReport`: cuando un centro de emergencias recibe de una ambulancia un `CloseMission` en el cual se informa de que un paciente ha fallecido durante el transporte a un hospital, este debe informar al hospital correspondiente mediante esta comunicación para que el hospital no siga esperando la llegada del paciente.

    - `MissionAssignement`: lo utilizan los centros de emergencias para solicitar a una ambulancia que proporcione asistencia medica a un paciente.

    - `ResponsePatient`: cuando un paciente llama a un centro de emergencias se le debe responder mediante este tipo de comunicación en la cual se le informará de que debe esperar a una ambulancia.

    - `ResponseMissionAssignement`: cuando una ambulancia recibe un `MissionAssignement` utilizará este tipo de comunicación para informar al centro de emergencias de su aceptación o no.

    - `HospitalRequest`: este tipo de comunicación lo utilizan las ambulancias para solicitar a un hospital la reserva de una cama para el paciente

al que esta atendiendo. Se utiliza para que el hospital pueda gestionar las camas que tiene disponible en base a las solicitudes que recibe.

- `CloseMission`: lo utilizan las ambulancias para informar al centro de emergencias de la finalización de una misión indicando si se ha realizado con éxito o no.
- `ResponseHospitalRequest`: una vez que un hospital recibe un `HospitalRequest` de una ambulancia, este le responderá con este tipo de mensaje indicando si acepta la petición o no.
- `CallEmergencyCenter`: sirve para que un paciente solicite a un centro de emergencias asistencia medica.

3.5.1.1.3. Ontología

En la figura 3.16 podemos observar los conceptos que se intercambian los agentes en el sistema, estos conceptos sirven para componer percepciones y acciones, e incluso algunos componen parte del estado interno de un agente. Para entender el significado de los conceptos especificados en la ontología estos serán analizados por cada tipo de agente:

- Paciente: si analizamos la percepción de un paciente podemos ver que se compone de una localización `TLocation`, de `TEmergencyState` que indica en que estado de una mision de emergencia se encuentra el paciente y de un estado físico `TPhysicalState` que determina la salud de un paciente a través de la cual se dará cuenta de si necesita asistencia medica o no. Cuando un agente quiere utilizar información relativa a un paciente utiliza el concepto `TInfoPatient`.
- Centro de emergencias: en su percepción se encuentra la información de todas las ambulancias `TInfoAmbulance`.

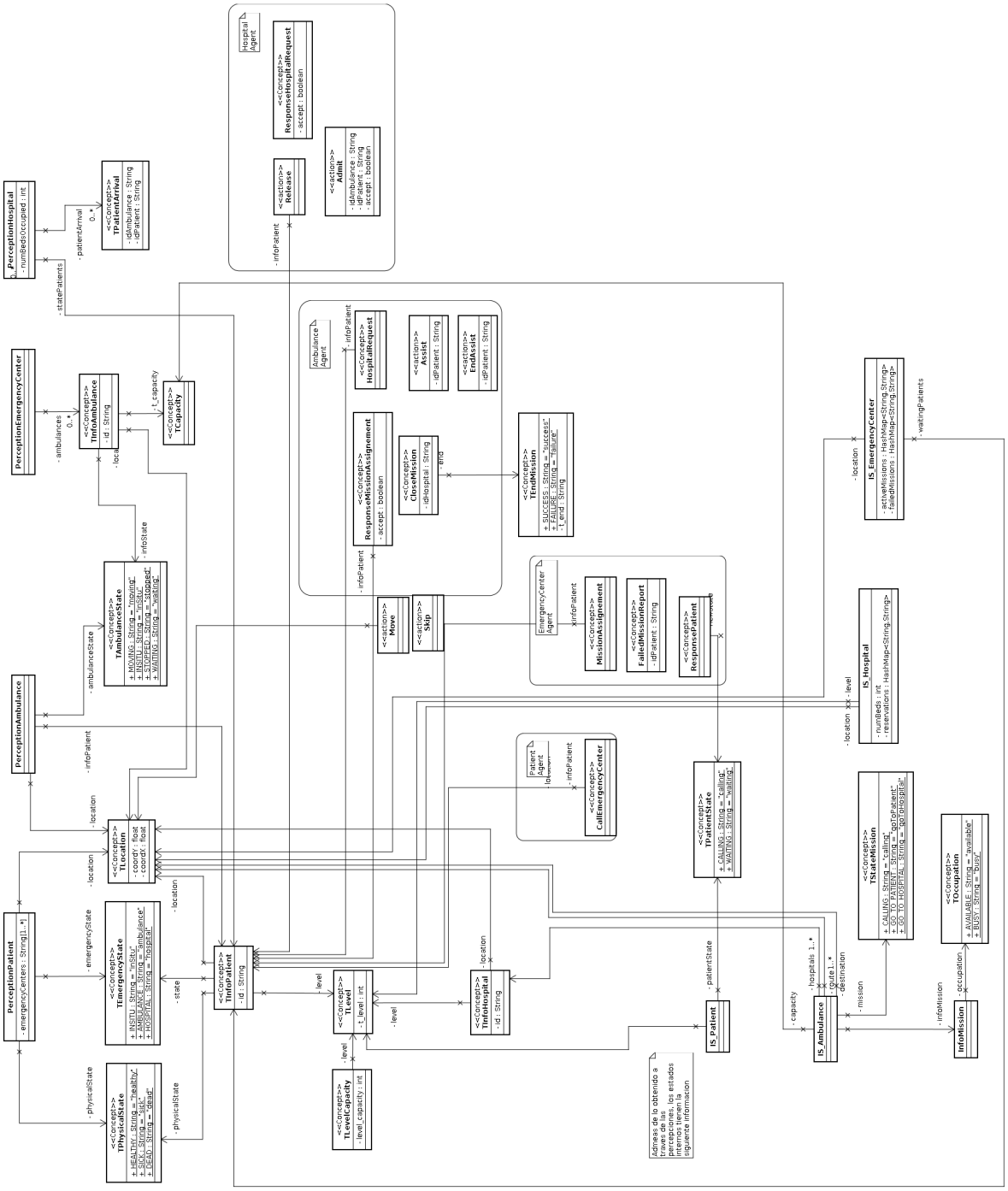


Figura 3.16: Ontología

- Ambulancia: para saber que esta realizando un ambulancia en cierto instante tenemos el concepto `TAmbulanceState`. El concepto `TInfoAmbulance` contiene toda la información necesaria para que otros agentes puedan referirse a cierta ambulancia.

- Hospital: de la percepción de los hospitales cabe destacar la llegada de pacientes mediante el concepto `TPatientArrival`. Para que los agentes puedan referirse a cierto hospital se utiliza el concepto `TInfoHospital`.

Ademas de los conceptos analizados disponemos de otro tipo de conceptos tales como `TCapacity` relativo a la capacidad de una ambulancia u hospital para tratar cierto nivel de emergencia `TLevel`, o `TEndMission` el cual utiliza una ambulancia dentro de una comunicación `CloseMission` para informar del resultado de la misión.

3.5.1.2. Simulador

El subsistema *Simulador* es el encargado de emular un escenario de emergencias médicas. Este subsistema se compone de un módulo denominado *Setup*, el cual permite al usuario configurar un escenario de urgencias médicas, un entorno, *Environment*, en el que actúan las diferentes entidades que participan en el escenario y un simulador, encargado de emular el escenario configurado a través de ese entorno y de la interacción con la *Aplicación*. Las clases que forman parte de este subsistema se describen en los siguientes subapartados:

3.5.1.2.1. Diagrama general de clases de la simulación

De forma global, se puede ver en el diagrama de clases 3.17 la estructura del módulo `Simulator`. El *Simulador* será iniciado por la clase `Initiator`, la cual creará una instancia de la clase SimulationParameters con toda la información

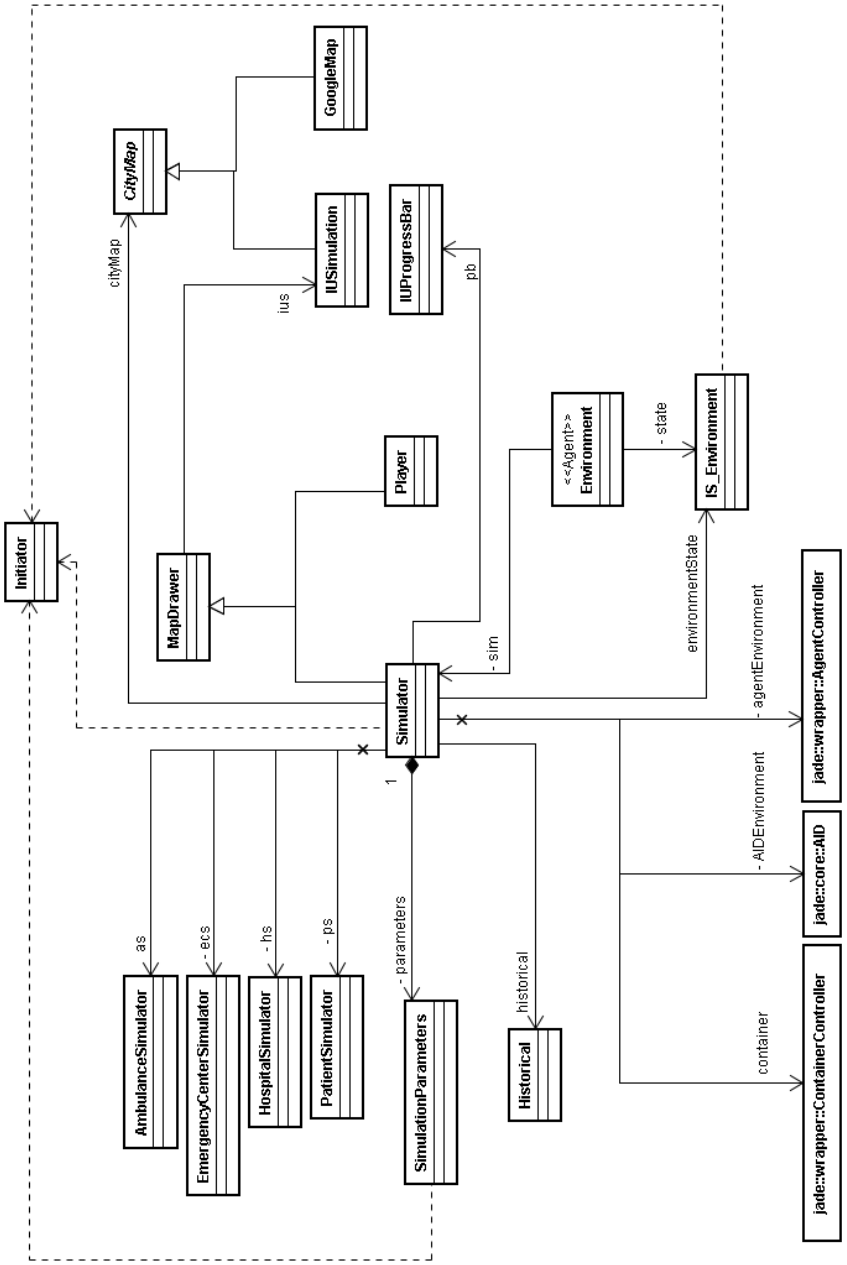


Figura 3.17: Diagrama general de clases de la simulación

necesaria para que la simulación se genere. Además, `Initiator` será quien cree el primer estado interno del entorno, el cual usará la clase `Simulator` para la creación de los diferentes agentes de la *Aplicación*. Esta secuencia de acciones se

puede ver con más detalle en el diagrama 3.31 del apartado de Diseño.

`Simulator` tendrá un contenedor de agentes llamado "container", donde se irán registrando todos los agentes creados. `Simulator` también guardará el AID (identificador de un agente) del entorno, para decir a cada agente ambulancia, paciente, hospital y centro coordinador con qué agente entorno debe interactuar.

El motor de simulación se subdivide en cuatro simuladores, uno por cada tipo de agente que hay en la *Aplicación*. Cada uno de ellos se encarga de simular las acciones que el agente entorno envía a `Simulator`, devolviendo a éste un nuevo estado del entorno, que a su vez `Simulator` devolverá al entorno. Todas estas actualizaciones del estado interno, junto con las acciones y comunicaciones de los agentes, se guardarán en la clase `Historical`.

Tanto `Simulator` como `Player`, son clases hijas de la clase `MapDrawer`. Ambas se encargan de ir enviando información sobre la simulación a la clase `IUSimulation` para que ésta la haga visible al usuario. La diferencia entre `Simulator` y `Player`, es que la primera clase visualiza una simulación en tiempo real y la segunda lo hace leyendo un fichero histórico elegido por el usuario.

Cuando se hace una simulación en tiempo real, se da al usuario la opción de elegir si quiere ver "on-line" los datos o guardarlos simplemente en un fichero. Si elige esta última opción, en vez de llamarse desde *Simulador* a `IUSimulation`, se llamará a `IUProgressBar`.

Una parte muy importante del *Simulador* es el cálculo de rutas y movimientos de los agentes. Para ello, se hace uso de la clase abstracta `CityMap` para calcular localizaciones de los agentes. En este demostrador se hace uso de la API de GoogleMaps para ese cálculo de localizaciones, API utilizada tanto por la clase `IUSimulation` como por la clase GoogleMap. La diferencia es que la primera se usará cuando la simulación se quiera visualizar al mismo tiempo que se genera, y la segunda clase se usará cuando la simulación se quiera realizar sin visualizado

inmediato. De esta forma, nos ahorramos la carga de dos navegadores Web al mismo tiempo (cada una de las clases contiene uno).

3.5.1.2.2. Diagrama general de clases del entorno

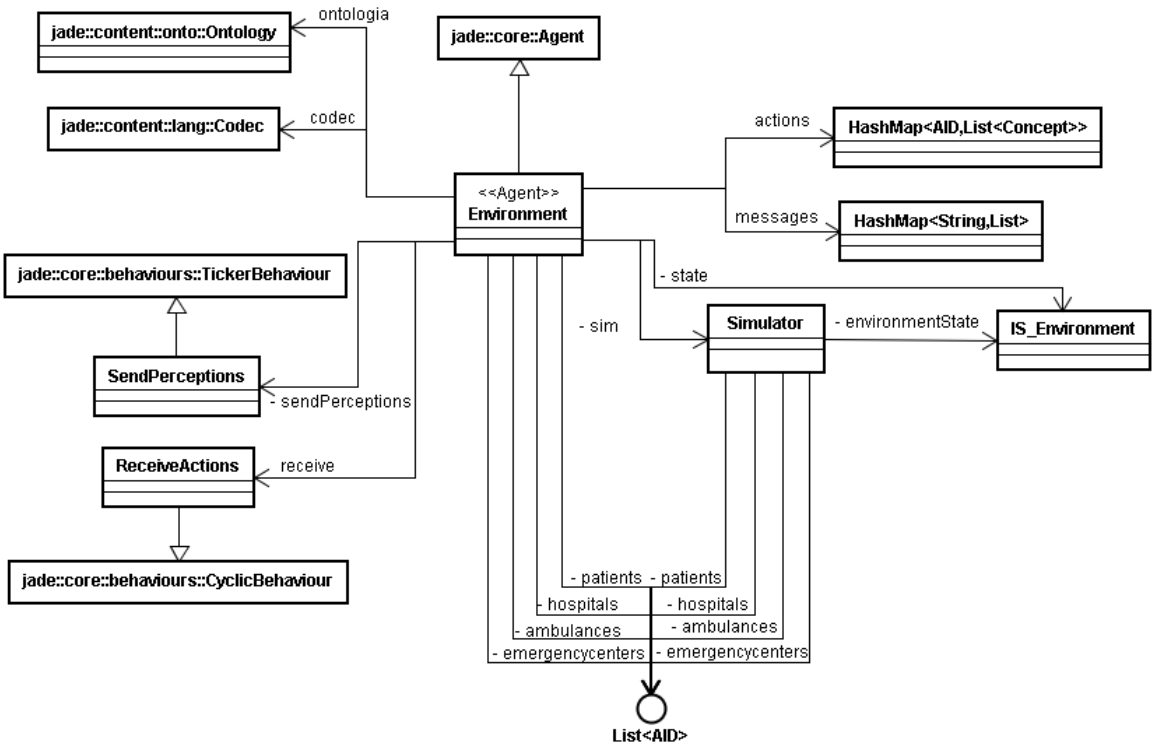


Figura 3.18: Diagrama general de clases del entorno

El agente entorno, mostrado en el diagrama 3.18, hace uso de diferentes clases para llevar a cabo sus principales funciones: enviar percepciones y recibir acciones de los agentes.

Este agente extiende a la clase `Agent` de JADE, dentro de ese método se registrará la ontología que usará el agente (`MHealthOntology` de la clase Ontology de JADE) y el Codec. El entorno recibirá su estado interno inicial, instancia de la clase ISEnvironment, y registrará un comportamiento para enviar percepciones

a los agentes que se encuentran en la simulación. Aunque en el estado interno el entorno puede ver los agentes del sistema, solo enviará percepciones a aquellos agentes que el *Simulador* le indique a través del método "setListAgents" ya que se le pasará por argumento el listado de AIDs de los agentes vivos en el sistema.

El comportamiento de enviar percepciones se implementará en la clase SendPerceptions, que hereda un comportamiento JADE TickerBehaviour. Este tipo de comportamiento lo que hace es que cada cierto periodo de tiempo, se ejecutará el código que contenga el método "onTick()". En nuestro entorno, en cada tick se crearán las percepciones de cada agente y se enviarán, a la vez que se registra un comportamiento para recibir las acciones de los agentes.

Este comportamiento, implementado en la clase ReceiveActions, hereda del comportamiento JADE CyclicBehaviour. Lo que hace es estar continuamente recibiendo acciones por parte de los demás agentes y almacenarlos en una cola de acciones y otra cola de comunicaciones. La cola de acciones será enviada al *Simulador* para que se simulen y se actualice el estado del entorno. Las comunicaciones se entregarán a los correspondientes receptores tal cual están.

Cada vez que se cumpla el periodo impuesto en el *tick* (el periodo será nuestro *timestep*), se eliminará el comportamiento cíclico de recibir acciones para que, mientras se envían las percepciones, no se reciban nuevas acciones, consiguiendo un comportamiento síncrono entre los agentes y el entorno.

3.5.1.2.3. Diagrama general de clases de la simulación de los agentes

Como ya se ha mencionado anteriormente, el motor de simulación, situado en la clase `Simulator`, se compone de unos parámetros de simulación y de unas listas con los AID de los agentes de la aplicación. El entorno recibe a través del simulador esa lista, para saber de quién recibirá acciones y a quienes tendrá que enviar percepciones. Como se ve en el diagrama 3.19, el simulador se

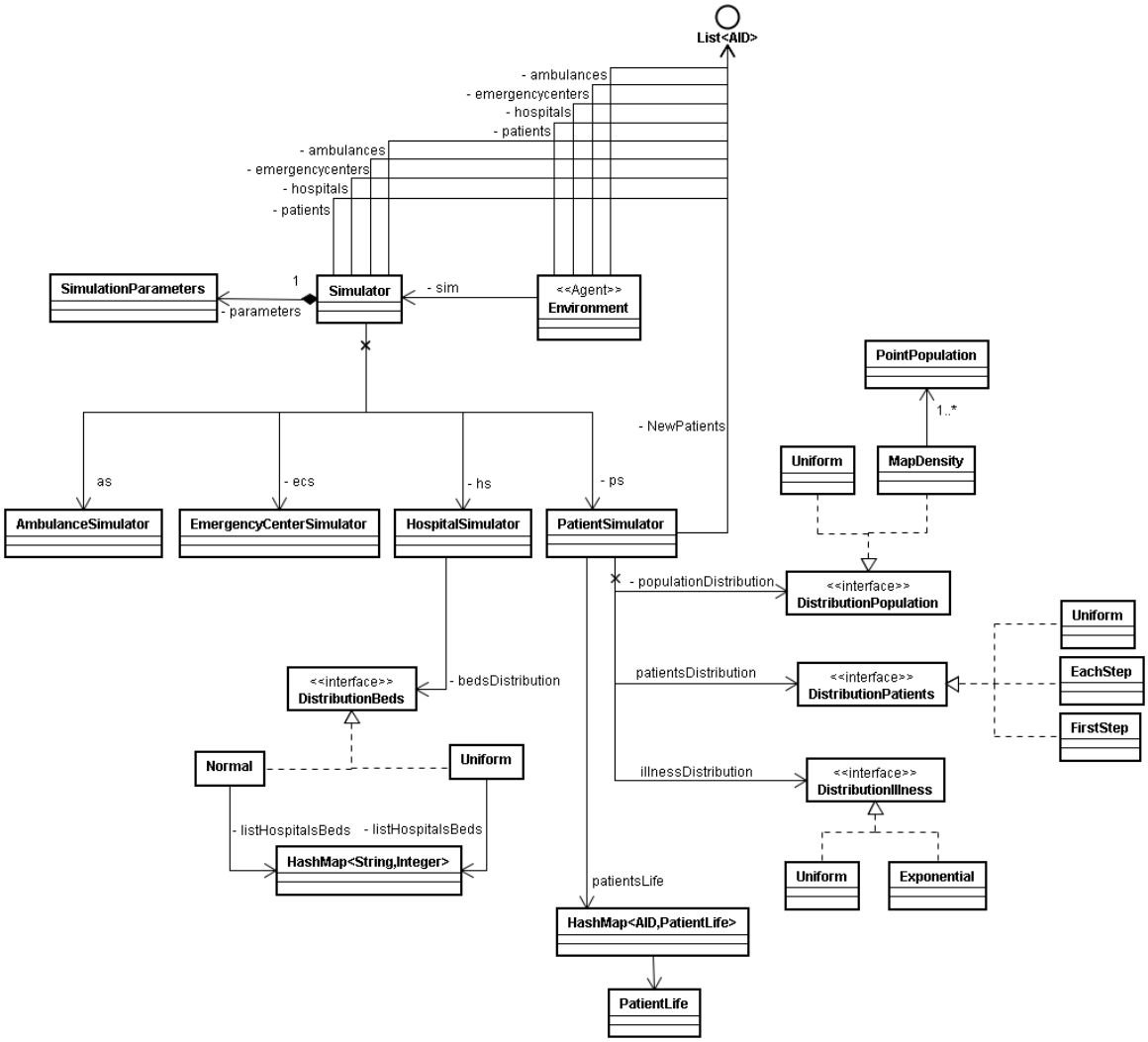


Figura 3.19: Diagrama general de clases de la simulación de los agentes

subdivide en cuatro clases `Simulator`, una por tipo de agente. Las acciones que lleguen al entorno serán enviadas a `Simulator` y éste comprobará que las acciones provienen de agentes que están en sus listas. Una vez hecha la comprobación, las clasificará según provengan de agentes ambulancias, centros de emergencia, hospitales o pacientes.

El simulador de ambulancias será el encargado de crear los agentes ambulancias, con los parámetros necesarios que recibirá de `Simulator`. Después, de-

berá simular cada acción que le llegue de cada ambulancia, devolviendo un nuevo estado del entorno con los cambios producidos en consecuencia de la realización de la acción. Hará uso del `CityMap` de `Simulator` para calcular localizaciones de las ambulancias, en el caso de que la acción recibida sea "Move". Para el resto de acciones, no necesita usar ninguna clase adicional.

El simulador de centros de emergencia inicializa los agentes de este tipo, enviándoles la información inicial apropiada. La única acción que puede realizar un centro de emergencias es "Skip", por lo que no se modificará el estado del entorno.

El simulador de hospitales, además de crear los agentes, se encarga de simular las acciones que estos quieren realizar en el entorno. Los hospitales tienen una parte aleatoria en la cual la ocupación de sus camas se va modificando durante la vida del agente, ya que se considera que el hospital puede recibir otros pacientes además de los que le llegan a través de las ambulancias. Para simular este efecto, se usan las clases `Normal` y `Uniform` de la interfaz `DistributionBeds`, las cuales irán modificando la ocupación de cada hospital según la distribución estadística que tienen implementadas internamente.

Finalmente, el simulador de pacientes. Los pacientes se van a ir añadiendo al sistema de forma aleatoria, basándose en diferentes distribuciones implementadas a través de la interfaz `DistributionPatients`. Con la clase `Uniform`, se generarán pacientes en el entorno de forma uniforme a lo largo de la simulación, con `EachStep` se creará un paciente en cada *timestep* desde el comienzo de la simulación y con `FirstStep`, se generarán todos los pacientes de golpe en el *timestep* inicial. Cuando se crean pacientes, éstos deben tener un nivel de enfermedad, el cual se distribuye con la interfaz `DistributionIllness`. Si se generan las enfermedades con `Uniform`, todos los niveles de enfermedad tendrán la misma probabilidad de ser seleccionados, mientras que con `Exponential`, se conseguirá que los niveles más bajos (más leves) de enfermedad sean más frecuentes que los niveles

más altos (más graves). Una vez se tenga el nivel de enfermedad de un paciente, es necesario ver dónde está situado en el mapa. Con `DistributionPopulation` se implementan dos tipos de distribuciones: `Uniform`, para que el paciente se coloque en cualquier parte de nuestro mapa de actuación y `MapDensity`, para que el paciente se sitúe en aquellos puntos con alta densidad de población con más probabilidad (estos datos provendrán del módulo *Setup*). Una vez creados los agentes, el `PatientSimulator` podrá simular las acciones que reciba de los pacientes,en nuestro sistema, solo `Move` (usando el `CityMap` de *Simulator*) y `Skip`. Tras la simulación de sus acciones, se procesará la vida y muerte de cada paciente, ya que cada agente tendrá asociado un `PatientLife` donde se refleja el tiempo que le queda para morir y los puntos de vida que se le van otorgando con cada asistencia médica, posibilitando así la opción de curarse.

3.5.1.2.4. Diagrama general de clases del histórico

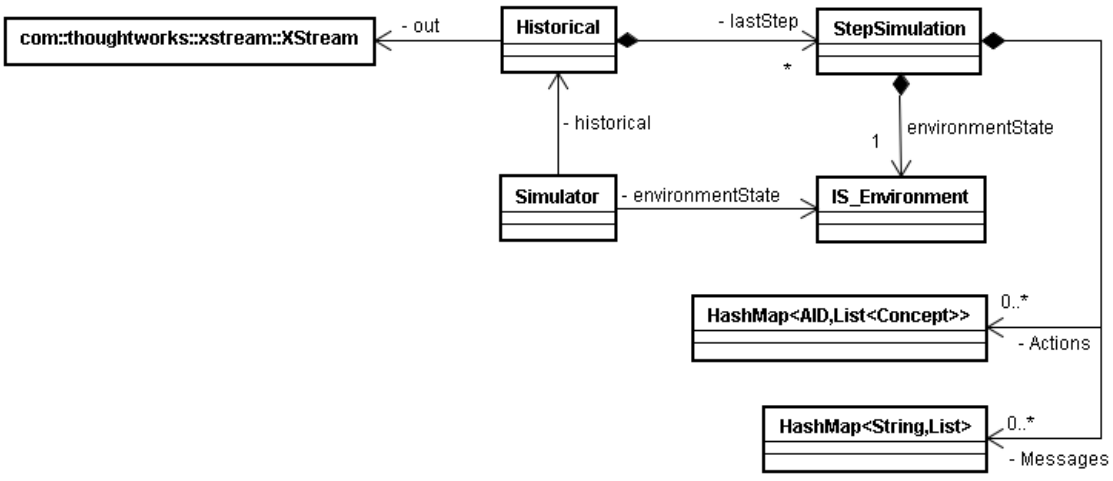


Figura 3.20: Diagrama general de clases del histórico

El proceso para guardar un histórico de la simulación se compone de las clases mostradas en el diagrama 3.20. *Simulator* tendrá en cada *timestep* un

estado del entorno actualizado. Cuando llame al método ***addNewStep()*** de la clase `Historical`, un nuevo paso de la simulación será creado. Ese paso, instancia de la clase `StepSimulation`, se compondrá del estado del entorno que actualmente maneja *Simulator*, junto con la lista de acciones y mensajes que los agentes enviaron al agente entorno. Cada paso de simulación se guardará en el fichero histórico en formato XML.

3.5.1.2.5. Diagrama general de clases de la simulación de mapas

El diagrama 3.21 muestra todas las clases implicadas en la simulación de rutas y movimientos de agentes. El navegador Web que cargará el fichero *GoogleMaps-Simulator.htm* se comunicará con las clases `GoogleMap` e `IUSimulation` a través de `MapLocation` (hereda de `BrowserFunction` para que haya comunicación bidireccional entre Java y JavaScript, tal y como se explica en el diagrama 3.59) del apartado de implementación. Tanto `GoogleMap` como `IUSimulation` pedirán al *Browser* que ejecute la acción de simular un movimiento desde un origen hasta un destino, y será un método JavaScript quien vaya actualizando las listas de *mapLocations* de cada clase, para que se sepa en cada momento la localización actual de cada agente. A su vez, ambas clases tendrán otra lista de *mapDestinys* para saber los destinos elegidos por los agentes, por si solicitan uno nuevo, eliminar la ruta previa y pedir al *Browser* que simule la nueva.

Tanto `IUSimulation` como `GoogleMaps` heredan de `CityMap`, y `Simulator` contiene una instancia de `CityMap`. Usará una clase u otra según se quiera una visualización de la simulación "on-line" u "off-line": para el primer caso se utilizará `IUSimulation`, ya que contiene todo lo necesario para ver gráficamente el desarrollo de la simulación y para el segundo caso se usará `GoogleMaps`, de forma que se calculen las rutas sin que el usuario tenga que ver el mapa que se está usando.

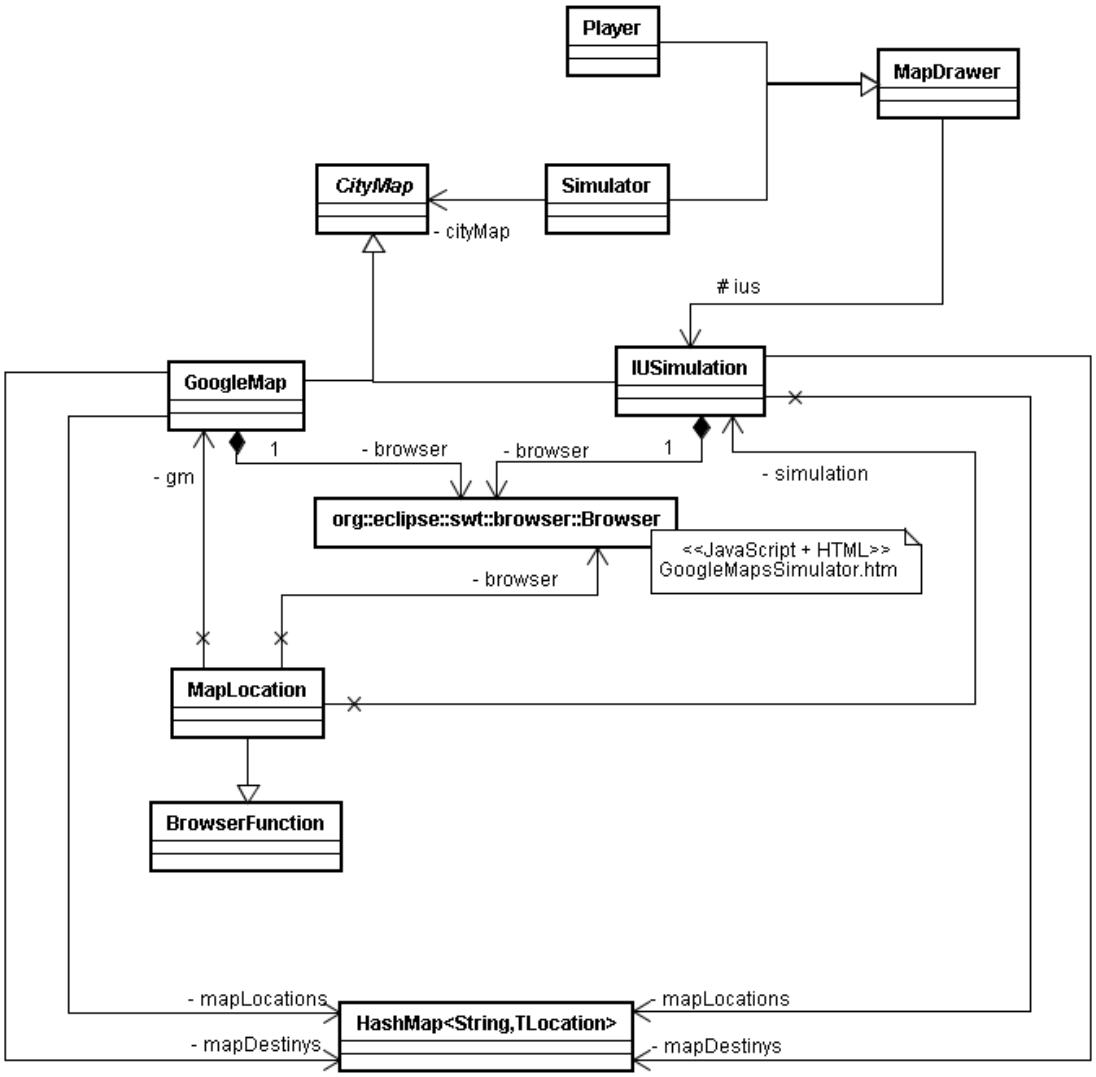


Figura 3.21: Diagrama general de clases del paquete Maps

3.5.1.2.6. Diagrama de clases del Setup

El diagrama 3.22 muestra la relación entre las clases del paquete *Setup*, el cual contiene la clase `IUSetupIndex` con el método ***`Main`*** que inicializará el programa. La función principal del *Setup* es mostrar al usuario las tres posibles acciones que puede realizar (cada una es uno de los casos de uso del usuario): crear configu-

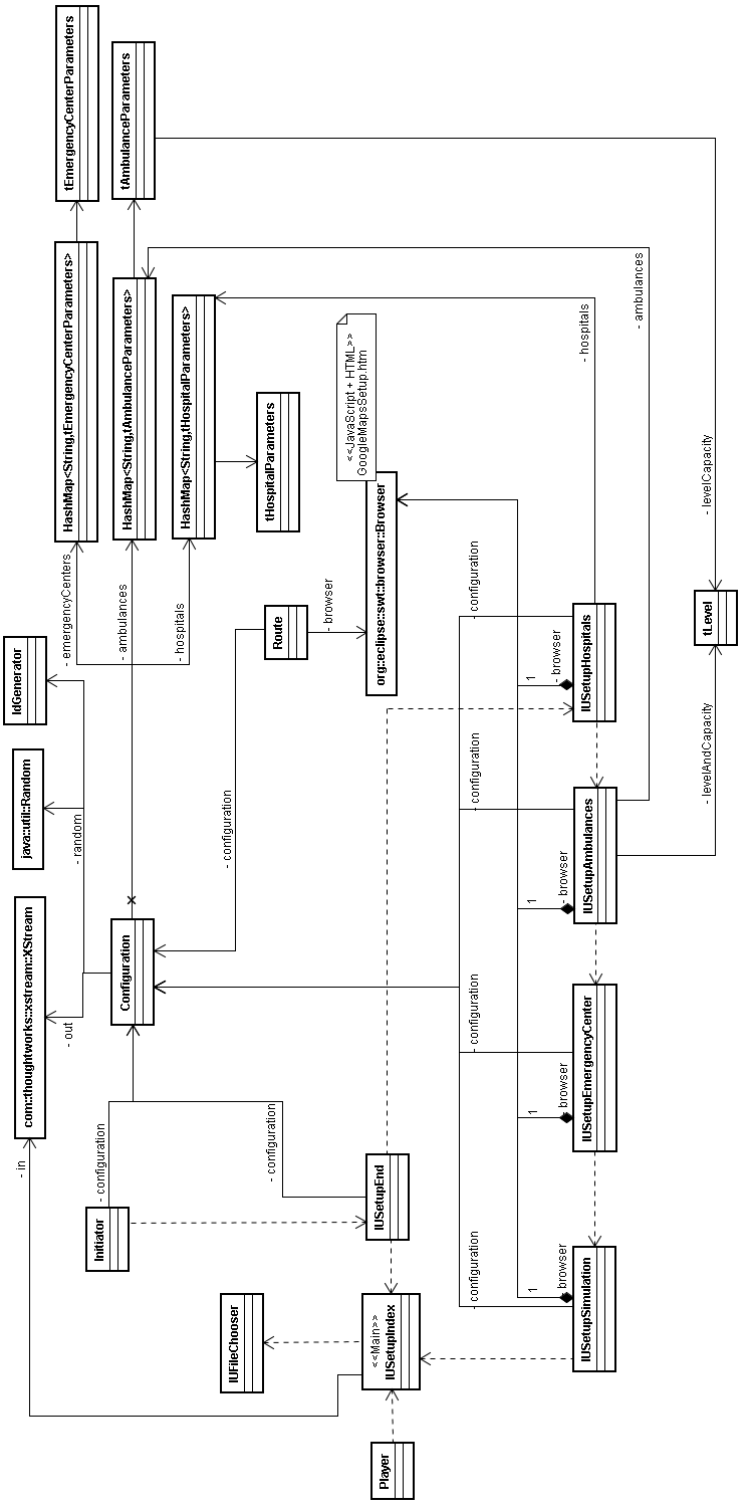


Figura 3.22: Diagrama general de clases del *Setup*

ración, realizar una simulación y visualizar una simulación.

En el primer caso, se mostrará al usuario gráficamente los parámetros de la simulación que se pueden configurar para que éste elija sus valores. Las clases implicadas en esta acción son las clases visuales `IUSetupSimulation`, `IUSetupEmergencyCenter`, `IUSetupAmbulances` e `IUSetupHospitals` (siguiendo este orden de aparición al usuario). Todas ellas se componen de un navegador SWT, que es el encargado de mostrar el fichero HTML *GoogleMapsSetup*, el cual contiene un mapa creado con la API de GoogleMaps y métodos JavaScript que manipulan ese mapa. En todas las clases visuales se recogen datos específicos de la simulación y se van almacenando en un objeto de la clase Configuration. Cuando se han introducido todos los parámetros, se muestra la clase `IUSetupEnd` donde se informará al usuario de dónde se van a guardar los ficheros de configuración y del histórico de la simulación. Finalmente, esta clase llamará al método *`StoreFile()`* de `Configuration` para que se cree un fichero XML con todos los datos.

Tanto si el usuario continúa con la simulación de la configuración creada como si carga en el sistema otra creada previamente, la clase `Initiator` se encargará de leer todos los parámetros de simulación para iniciar la ejecución.

En el caso de que el usuario elija en `IUSetupIndex` la opción de visualización, se leerá un fichero XML con el histórico de una simulación previa y la clase `Player` será la encargada de interactuar con la interfaz del módulo *Visualizador* para que se vayan mostrando todos los pasos de la simulación.

3.5.1.3. Visualizador

La funcionalidad del módulo *Visualizador* es mostrar gráficamente la información de la simulación en cada *timestep*. Como se aprecia en el diagrama 3.23, tanto `Simulator` como `Player` usan la clase visual `IUSimulation` que tiene su padre, `MapDrawer`.

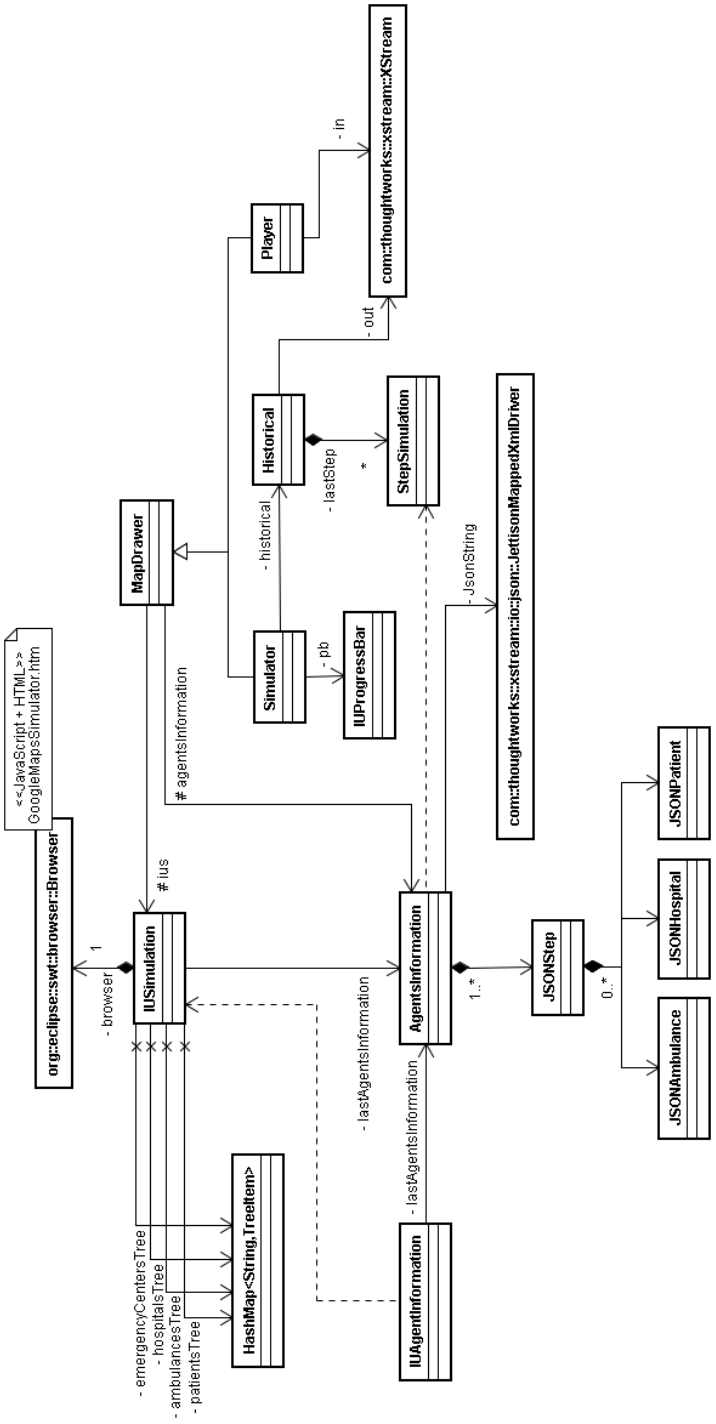


Figura 3.23: Diagrama general de clases del módulo *Visualizador*

`Player` siempre mostrará la información que lea del fichero histórico seleccionado por el usuario llamando en cada *timestep* al método ***`DrawStep()`*** de `IUSimulation`. *Simulator* podrá usar la clase `IUProgressBar` para indicar al usuario el progreso de la simulación en el caso de que éste no quiera visualizar nada, solo quiera almacenar datos, o mostrar la clase `IUSimulation` con toda la información gráfica en tiempo real.

El método ***`DrawStep()`*** recibe una instancia de la clase `AgentsInformation`. Esta clase se encarga de procesar toda la información del entorno a través de cada `StepSimulation`. Esta información se verá gráficamente de diferentes formas: la primera es a través de árboles de información en `IUSimulation`, los cuales tendrán los identificadores de cada agente con un *String* resumiendo su estado actual. La segunda forma es cuando se hace *click* sobre cada una de las ramas de esos árboles: se abrirán ventanas de información `IUAgentsInformation` con más datos sobre el agente, así como un histórico de sus acciones. Y la tercera y última forma es a través de un navegador Web, el cual cargará el fichero *GoogleMaps-Simulator.htm* que mostrará la localización de cada agente en el mapa del entorno y pop-pus de información en cada icono representativo de los agentes. Para que el navegador sepa interpretar esa información, se usan las clases `JSONAmbulance`, `JSONHospital` y `JSONPatient` para enviar en un *String* en formato JSON los datos al método JavaScript de visualización.

Finalmente, cuando ésta visualización acabe, se volverá al módulo *Setup* para continuar con más simulaciones, cargar históricos o crear más configuraciones.

3.5.2. Diagramas de secuencia por caso de uso

Para comprender mejor cómo colaboran los distintos objetos implicados en cada caso de uso, se muestran en esta sección una serie de diagramas de interacción, los cuales sirven para ver cronológicamente la secuencia de interacciones

que se producen en cada caso de uso de nuestro proyecto.

Cada caso de uso se detalla con su camino básico y los caminos alternativos, es decir, su proceso normal de ejecución y los procesos secundarios que se puedan dar, si proceden. A modo de ejemplo, en cada caso de uso se muestra el diagrama de secuencia del camino básico, pudiendo ver el resto de diagramas en el Anexo B de esta memoria.

3.5.2.1. Aplicación

En el subsistema *Aplicación* se encontrarán los actores del dominio de las emergencias médicas, estos utilizarán una variación del protocolo *request* definido en la sección de análisis 3.4.3 para la interacción con el entorno.

En esta sección mostraremos mediante diagramas de secuencia las interacciones entre los diferentes actores de la *Aplicación* que se realizan en el entorno del *Simulador*.

3.5.2.1.1. Percibir el entorno

La realización de este caso de uso esta implícita en todos los casos de uso que vamos a detallar en esta sección, esto se debe a la propia definición de agente racional, es decir, actúa en consecuencia a sus percepciones.

3.5.2.1.2. Comunicarse con otros agentes

Al igual que el caso de uso anterior este caso de uso esta implícito en la mayoría de los casos de usos, ya que para realizar muchos de ellos es necesaria una comunicación previa entre cierto tipo de agentes.

En particular, las comunicaciones existentes son las siguientes:

- Comunicación entre paciente y centro de emergencias: se realiza cuando el

paciente enferma y solicita asistencia medica a un centro de emergencias. Ver figura 3.24.

- Comunicación entre centro coordinador y ambulancia: esta comunicación se realiza en varias ocasiones: la primera de ellas es cuando el centro coordinador solicita una ambulancia para una asistencia medica y la segunda es cuando dicha ambulancia finaliza la asistencia medica.
- Comunicación entre ambulancia y hospital: se realiza cuando una ambulancia solicita acceso para un paciente a un hospital.

3.5.2.1.3. Moverse

Tanto las ambulancias como los pacientes tienen la capacidad de moverse, es decir, son capaces de ir de una localización a otra. Para conseguirlo es posible que necesiten realizar más de una acción `Move`, esto se debe a que el movimiento es simulado, por lo que las distancias influyen en la capacidad de movimiento de los agentes. En el diagrama 3.25 podemos observar como una ambulancia se mueve hasta la localización de un paciente.

3.5.2.1.4. Transportar pacientes

Las ambulancias son capaces de transportar pacientes a los hospitales, para la realización de este caso la ambulancia tiene que estar en una misión en la que ha asistido *in situ* a un paciente y no le ha conseguido curar, por lo que lo tiene que llevar a un hospital. En el diagrama 3.26 podemos observar como una ambulancia se trasporta a un paciente a un hospital y espera hasta que se realiza su admisión.

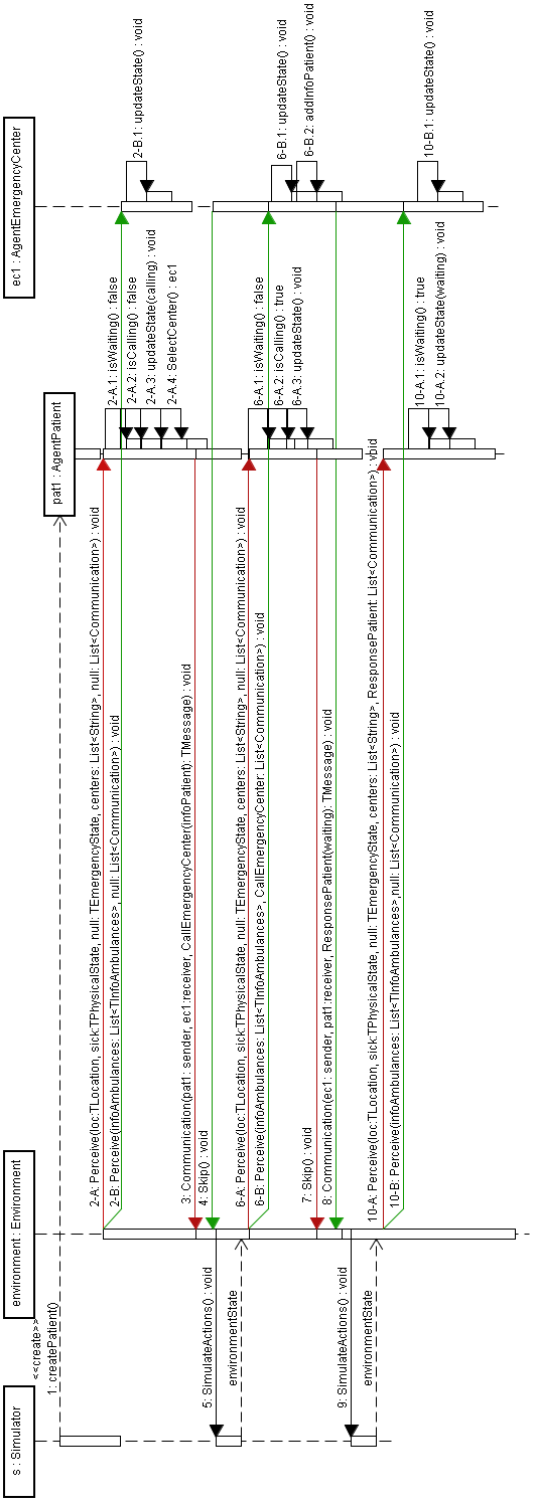


Figura 3.24: Diagrama de secuencia: Una paciente enferma y se pone en contacto con el centro de emergencias

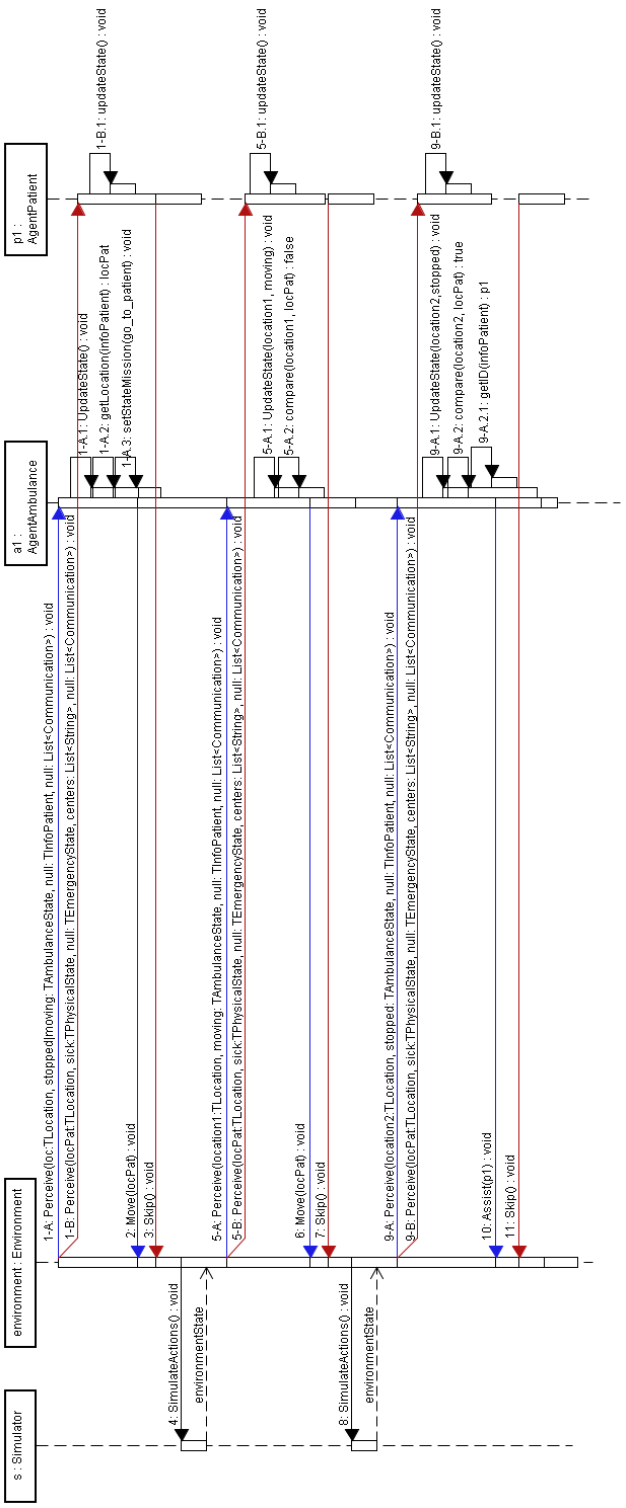


Figura 3.25: Diagrama de secuencia: Una ambulancia se dirige hacia un paciente

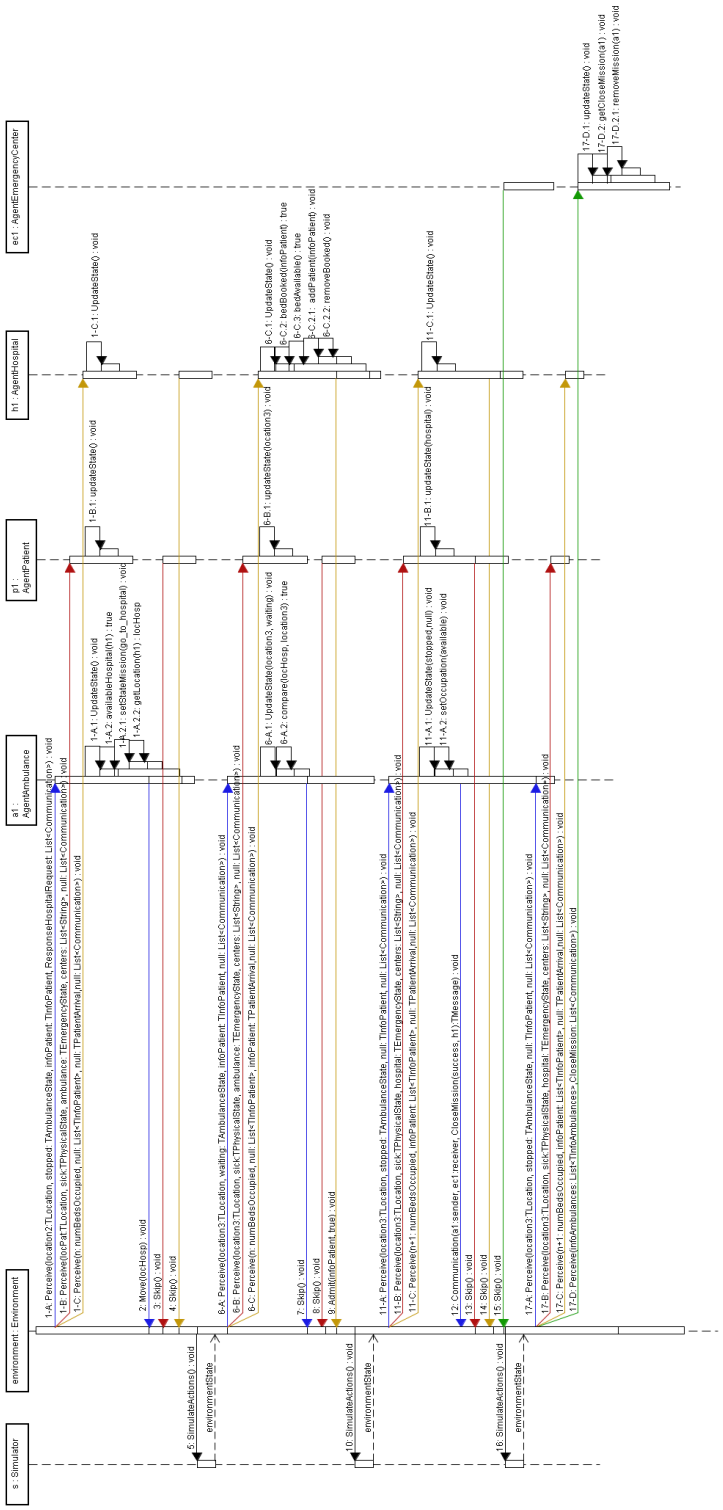


Figura 3.26: Diagrama de secuencia: Una ambulancia transporta un paciente a un hospital

3.5.2.1.5. Aceptar misiones de emergencias

Las ambulancias pueden aceptar misiones de emergencias en base a su disponibilidad. En el diagrama 3.27 observamos cómo una ambulancia que esta disponible se comunica con el centro de emergencias para la asignación de una misión.

3.5.2.1.6. Rechazar misiones de emergencias

Este caso de uso es similar al anterior, solo que en este caso la ambulancia rechazará la misión en el caso de que no este disponible. En el diagrama 3.28 se observa cómo un centro de emergencias contacta con una ambulancia para asignarla una misión, pero la ambulancia esta ocupada y, por lo tanto, hay que localizar otra ambulancia.

3.5.2.1.7. Asistir pacientes

Las ambulancias tienen la capacidad de asistir *in situ* a los pacientes, por ello realizarán tantas asistencias como crean necesarias para intentar curarlos, en caso de conseguirlo se finalizará la misión y en caso contrario el paciente será transportado a un hospital. En el diagrama 3.29 vemos la secuencia en la que la ambulancia es capaz de curar al paciente.

3.5.2.1.8. Aceptar admisiones de pacientes

Los hospitales son capaces de admitir pacientes en sus instalaciones para proporcionarles asistencia medica, para poder admitir a un paciente hay que cumplir dos condiciones, la primera y más restrictiva es que el hospital no este ocupado por completo y la segunda es la reserva previa que debe realizar una ambulancia para un paciente. De esta manera, una vez aceptada la reserva, cuando una ambulancia llega a la localización del hospital el hospital podrá admitir al paciente

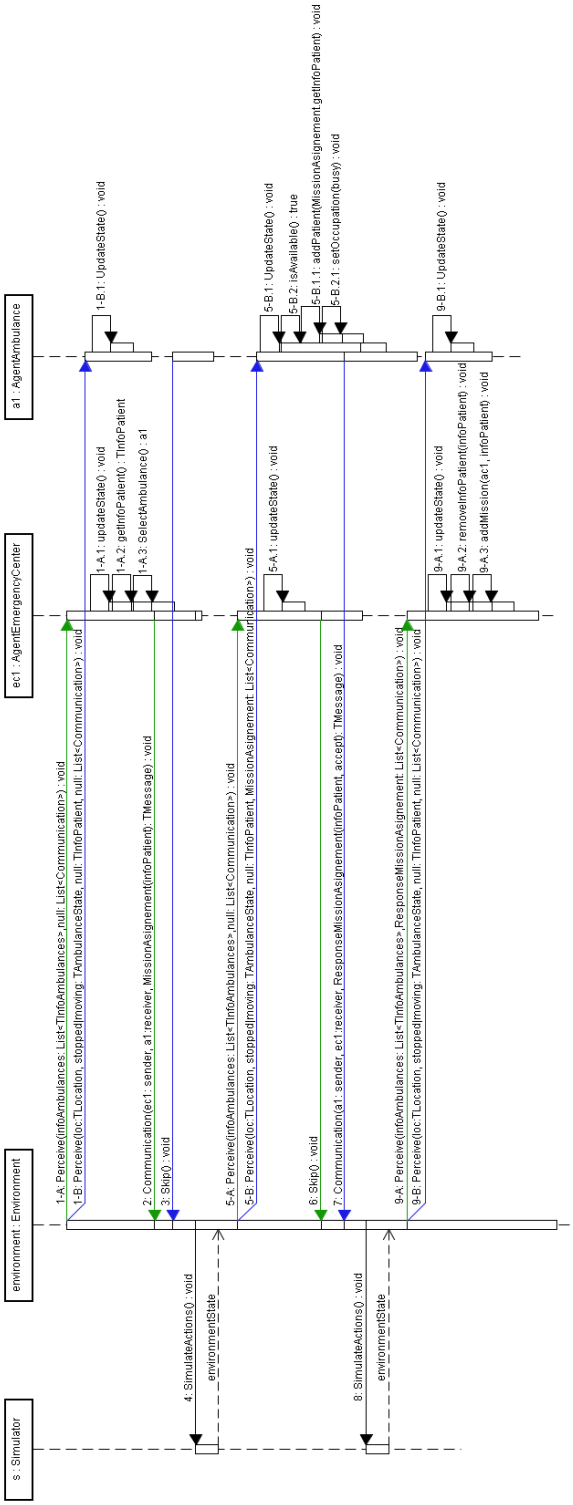


Figura 3.27: Diagrama de secuencia: Una ambulancia acepta una misión de emergencia

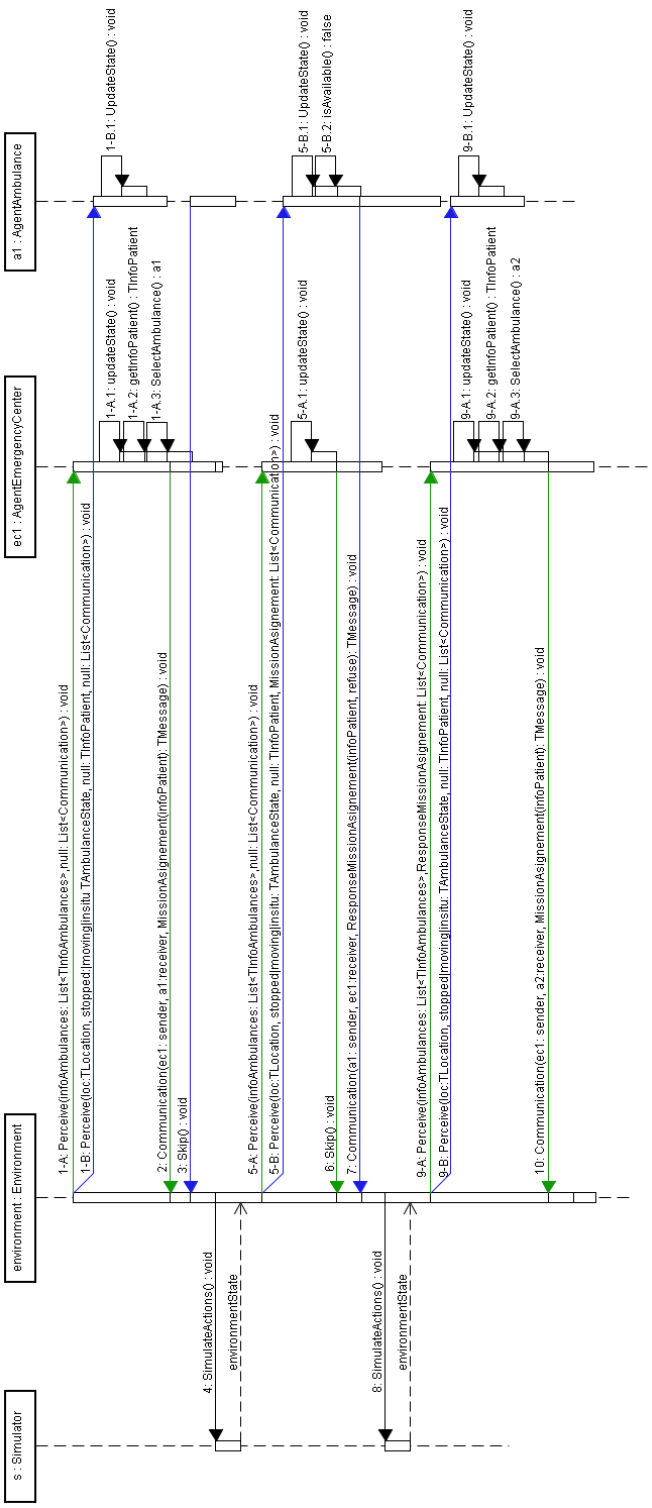


Figura 3.28: Diagrama de secuencia: Una ambulancia rechaza una misión de emergencia

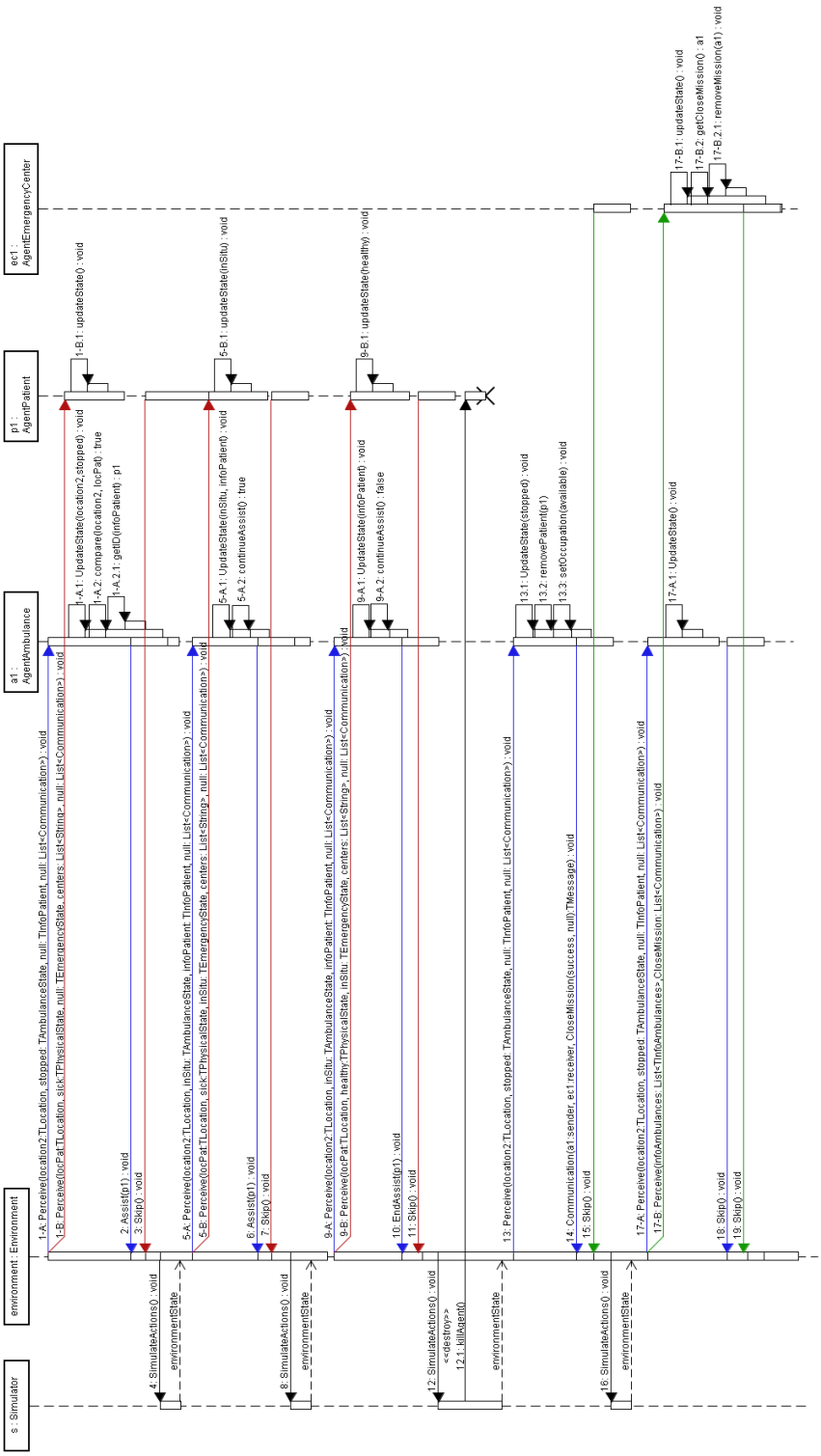


Figura 3.29: Diagrama de secuencia: Una ambulancia asiste a un paciente y le cura

siempre y cuando haya disponibilidad de camas en el mismo. Podemos ver esta interacción en el diagrama 3.26.

3.5.2.1.9. Rechazar admisiones de pacientes

Este caso de uso es similar al anterior solo que en este caso el hospital rechazara el acceso a un paciente por no haber realizado la reserva o no haber disponibilidad. Ver 3.30.

3.5.2.2. Simulador

El subsistema *Simulador* es el encargado de realizar las simulaciones, crear las configuraciones iniciales y guardar toda la información generada.

La secuencia de acciones que sigue la ejecución de una simulación, fue descrita en el diagrama de estados 3.6 , en la sección de Análisis. Este caso de uso, al ser tan complejo, se va a descomponer en varios diagramas de secuencia que juntos, completan la funcionalidad de ejecutar una simulación. El orden que sigue es el siguiente:

3.5.2.2.1. Leer fichero de configuración y crear el estado del entorno inicial

Camino Estándar: Este caso de uso comienza cuando el usuario selecciona la opción "Ejecutar una simulación" en la pantalla de inicio. A partir de ahí, tal y como se muestra en el diagrama 3.31, se le solicita al usuario que seleccione el fichero con la configuración de la simulación a realizar. Una vez leída, se le pregunta si desea visualizar la simulación mientras se realiza o no. Seleccione si o no, la simulación dará comienzo a través de la clase `Initiator`, que es la encargada de convertir la configuración leída del fichero a los parámetros necesarios en la clase `Simulator`. En esa conversión, se incluye la creación de un estado inicial del entorno a simular. Una vez que todos los parámetros han sido almacenados en la

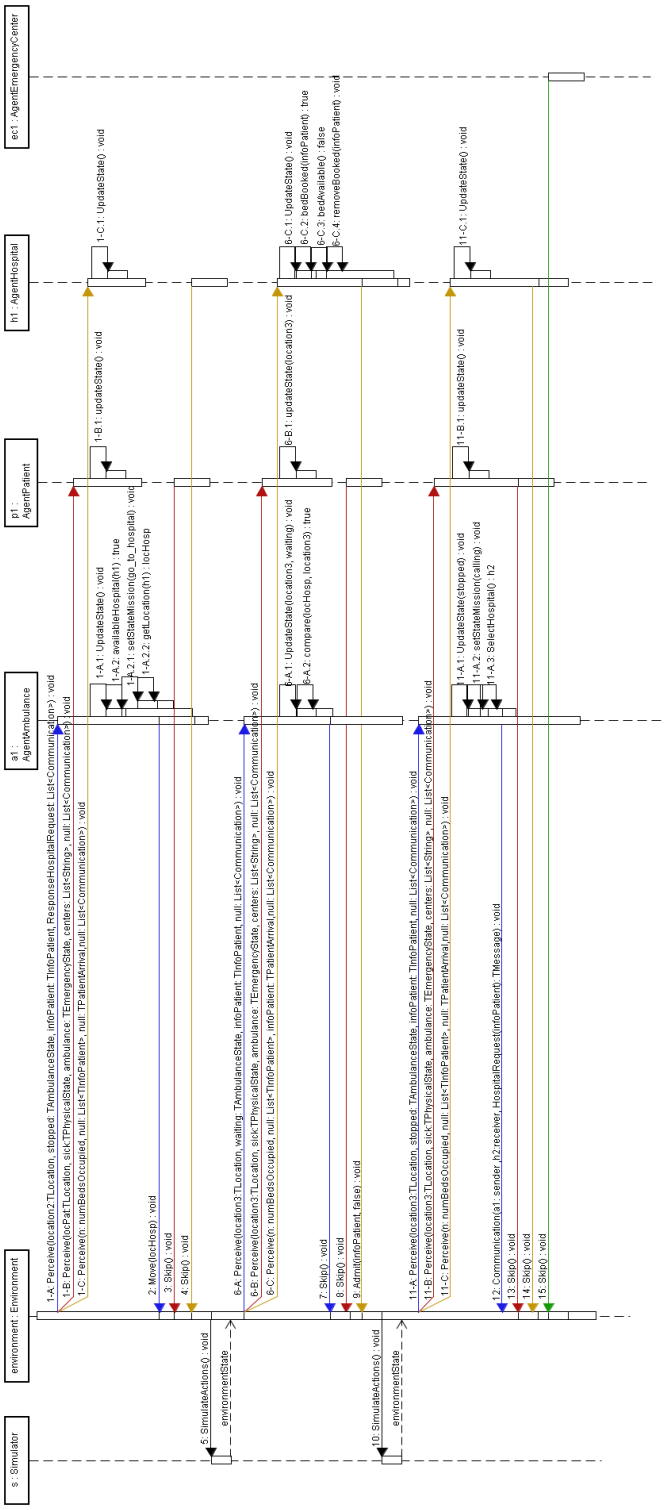


Figura 3.30: Diagrama de secuencia: Un hospital rechaza un paciente

Figura 3.31: Diagrama de secuencia: iniciar simulador

clase `SimulationParameters` y el estado del entorno ha sido creado, se lanza el simulador llamando al método *`start()`*.

Camino Alternativo 1: Si el usuario selecciona un fichero de configuración no válido, se le muestra un mensaje de error y se vuelve a la pantalla de inicio, para que empiece de nuevo si quiere.

3.5.2.2.2. Comenzar la simulación, creando los agentes y el entorno

Camino Estándar: En el diagrama de secuencia 3.32 se detalla el comienzo de una simulación. Cuando se instancia la clase `Simulator`, se inicializan otras clase que son simuladores más específicos de cada tipo de agente del entorno. En la creación de esos simuladores, se les envían los parámetros de simulación, ya que en ese objeto están guardados entre otros datos, los tipos de distribuciones estadísticas que tienen que usarse en la simulación. Además, se crea el fichero que almacenará el historial de simulación en cada *timestep*. Cuando se llama al método *`start()`* de `Simulator`, se guarda el primer paso de la simulación, el inicial, que consistirá en una lista de acciones y mensajes vacías y el estado del entorno creado por `Initiator`. Después, se crea el agente entorno, dentro de un contenedor de agentes, se le envía su estado interno y se le inicializa, para que se quede a la escucha de peticiones de acciones por parte del resto de agentes, que son creados nada más comience el entorno a funcionar. El hecho de que se creen después es porque necesitan conocer el identificador del entorno para poder comunicarse con él, ya que ese dato no se conoce a priori. Tras la creación de las ambulancias, hospitales y centros de emergencia, se envía al entorno una lista de los identificadores de estos para que sólo acepte las acciones provenientes de estos agentes. Al ser un estado inicial, no se crean pacientes en el entorno, por lo que se le envía una lista de pacientes vacía. Finalmente, se le pide a la clase `Historical` el último paso de simulación creado. En este momento, se corresponde con el

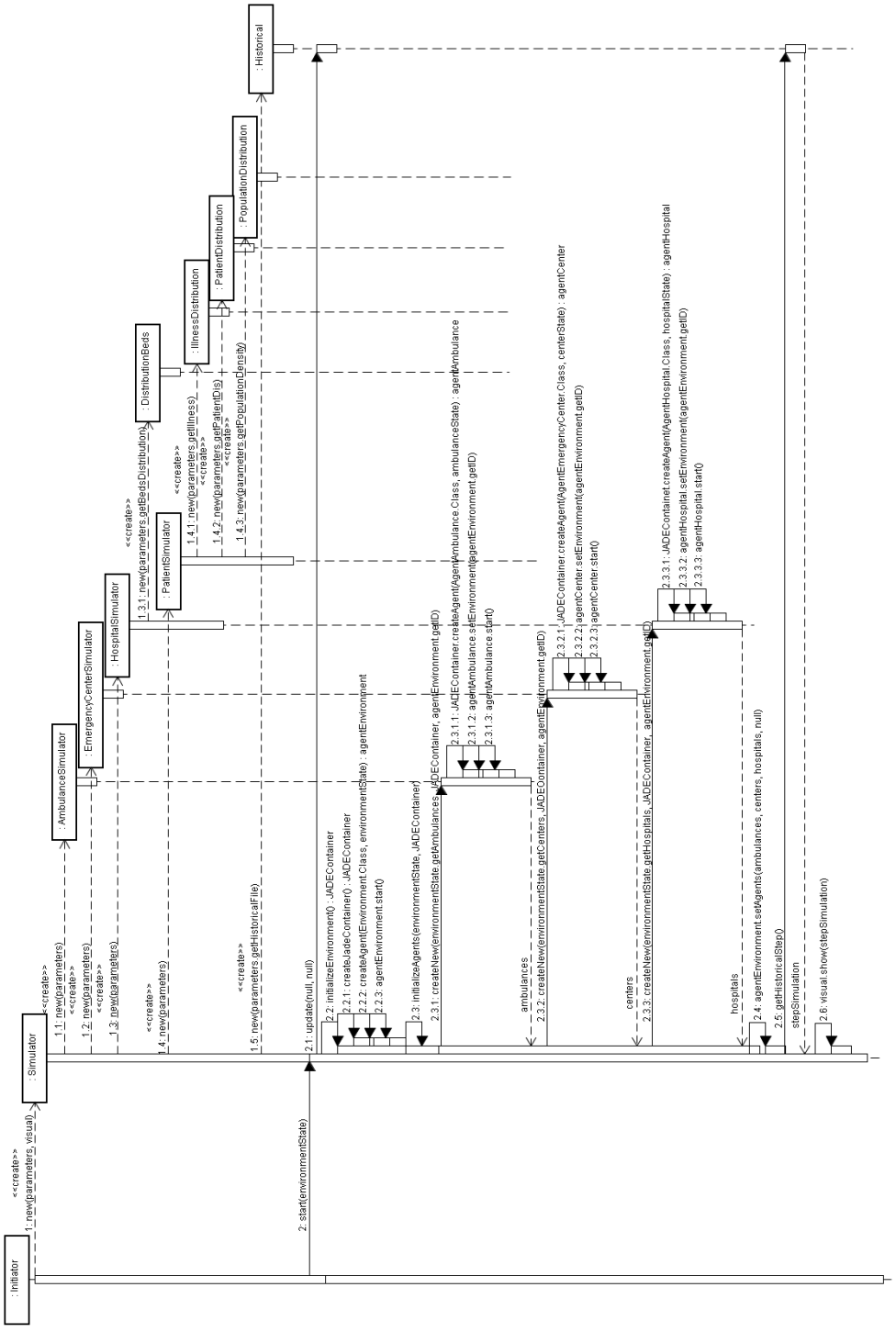


Figura 3.32: Diagrama de secuencia: creación del entorno y los agentes

estado inicial y estos datos se envían al subsistema *Visualizador* para que los muestre, en el caso de que el usuario seleccionase la opción de visualización al principio.

3.5.2.2.3. Simular acciones y enviar el nuevo estado al entorno

Una vez se ha preparado toda la simulación, la clase `Simulator` queda a la espera de que el agente Entorno le envíe acciones para simular. Esta simulación de acciones es la parte más compleja del subsistema *Simulador*. Cada acción que se solicite emular debe provenir de un agente registrado en el sistema, y la acción tendrá unos efectos en el entorno u otros, dependiendo del tipo de agente que ha enviado la acción y de cómo se encuentre el estado del entorno en ese momento.

Los pasos que sigue la simulación de acciones es, primero, emular las acciones que ha recibido el entorno de forma síncrona, una por una. Después, se procesa el estado de vida de los pacientes y finalmente, se procesa la ocupación de las camas de los hospitales. Estos dos últimos procesamientos, son los que dotan de aleatoriedad a nuestro entorno.

A continuación, se muestran los diagramas de secuencia clasificándose por tipo de agente emisor de la acción y después, los diagramas asociados al procesamiento de pacientes y hospitales:

- Diagramas de secuencia para simular las acciones de las ambulancias
  - Simular "Assist":

    Camino Estándar: En el diagrama de secuencia 3.33, se muestra la secuencia de pasos que se siguen para emular la asistencia de una ambulancia a un paciente. El camino básico consiste en que la ambulancia se encuentra en la misma localización física del paciente, percibe que este paciente está enfermo y no está siendo atendido por ninguna

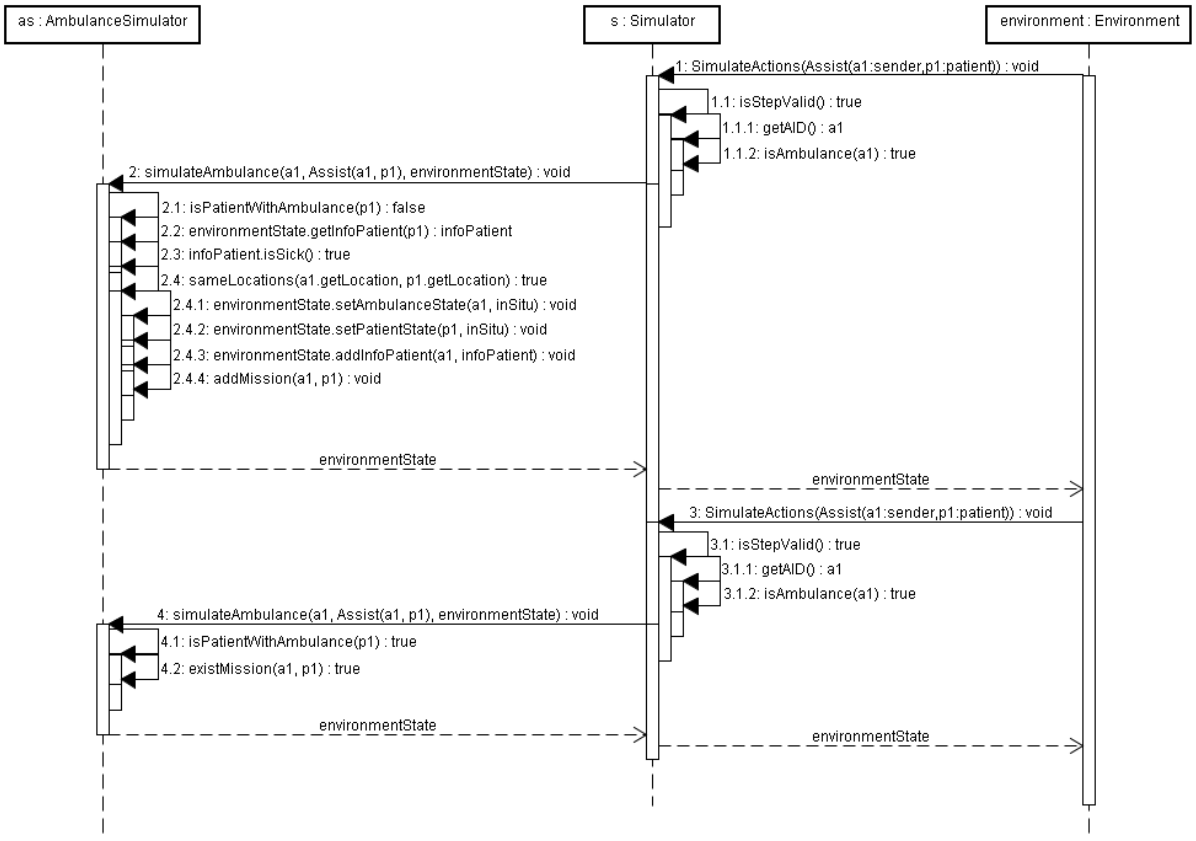


Figura 3.33: Diagrama de secuencia: Se simula que una ambulancia asiste a un paciente

otra ambulancia. Entonces, el paciente comienza a recibir asistencia, poniendo su estado interno a *in situ* y el de la ambulancia también, para que sepa que está ocupada con un paciente. Los datos de ese paciente se le envían a la ambulancia y se almacena en el simulador una misión activa entre ambulancia y paciente. Cuando se quiere hacer otra asistencia de una misión ya activa, no se modifica nada del entorno, debido a que los agentes implicados no modifican sus estados.

Camino Alternativo 1: el paciente ya no está enfermo mientras la ambulancia le está asistiendo. Puede tener el estado de enfermo o muerto. En ambos casos, la ambulancia recibe el estado *Stopped* puesto que no puede realizar una asistencia al paciente. De esta forma, y viendo que no recibe la información del paciente, sabrá que el paciente al que quiere asistir se encuentra curado o muerto. Ver diagrama B.2.5.2 del Anexo B.

Camino Alternativo 2: la ambulancia quiere asistir a un paciente que ya se encuentra en otra ambulancia. En ese caso, la ambulancia recibe el estado *Stopped*, para que sepa que la asistencia *in situ* no se pudo llevar a cabo. Ver diagrama B.2.5.3 del Anexo B.

Camino Alternativo 3: la ambulancia quiere asistir a un paciente que no se encuentra en la misma localización física que ella. Recibirá su estado de misión como *Stopped*, y así sabrá que no se pudo realizar la asistencia al paciente.Ver diagrama B.2.5.4 del Anexo B.

- Simular "EndAssist":

  Camino Estándar: en el diagrama 3.34, se detalla como la ambulancia realiza un cierre de asistencia, a pesar de que el paciente sigue enfermo. Para cerrar una misión, ésta debe existir previamente. La ambulancia tomará el estado *Stopped* pero seguirá percibiendo al pa-

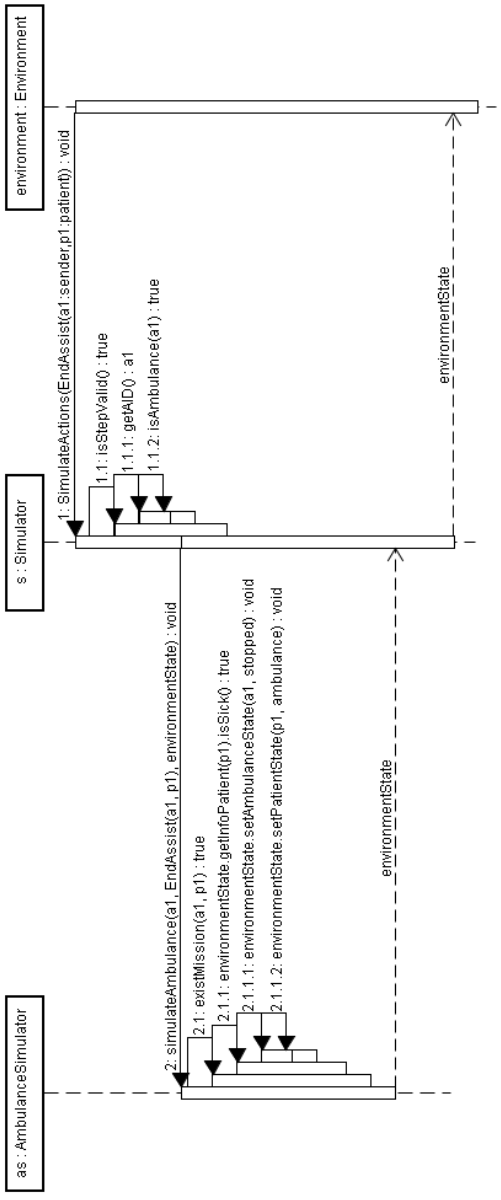


Figura 3.34: Diagrama de secuencia: Se simula el fin de asistencia de la ambulancia al paciente

ciente, ya que éste sigue en la ambulancia. Su estado será *Ambulance*.

Camino Alternativo 1: se realiza un cierre de asistencia porque el paciente deja de estar enfermo. Al curarse o morirse, la ambulancia le da un alta simbólico, poniendo su propio estado a *Stopped* y eliminando su percepción del paciente, pues ya no le tiene en su interior. El paciente no modifica su estado en esta acción, ya que si está vivo o muerto, su agente asociado deja de existir en el entorno. Ver diagrama B.2.6.2 del Anexo B.

Camino Alternativo 2: la ambulancia quiere hacer un cierre de asistencia de una misión que no existe. En ese caso, se pone su estado a *Stopped* y no se modifica nada más, ya que la ambulancia no se encontraba en ninguna misión activa. Ver diagrama B.2.6.3 del Anexo B.

- Simular "Move":

  Camino Estándar: en el diagrama 3.35,se ve como una ambulancia ha solicitado realizar un movimiento. Lo primero que hace el simulador, es mirar si la ambulancia tiene un paciente. En ese caso, sabe que es un traslado del paciente a un hospital. Se solicita a la clase que esté realizando los cálculos de movimientos, clase que implementa la interfaz CityMap, que calcule la nueva posición de la ambulancia. El resultado se le asigna a la ambulancia y al paciente que tiene dentro. Se comprueba si la nueva localización coincide con la localización destino de la ambulancia. Al ver que no son las mismas, la ambulancia se pone con el estado *Moving* y el paciente se mantiene en *Ambulance*. A los centros coordinadores se les notifica el nuevo estado y localización de la ambulancia, para que sepan donde están las ambulancias en cada momento. En la siguiente iteracion, se puede ver cómo las localizaciones

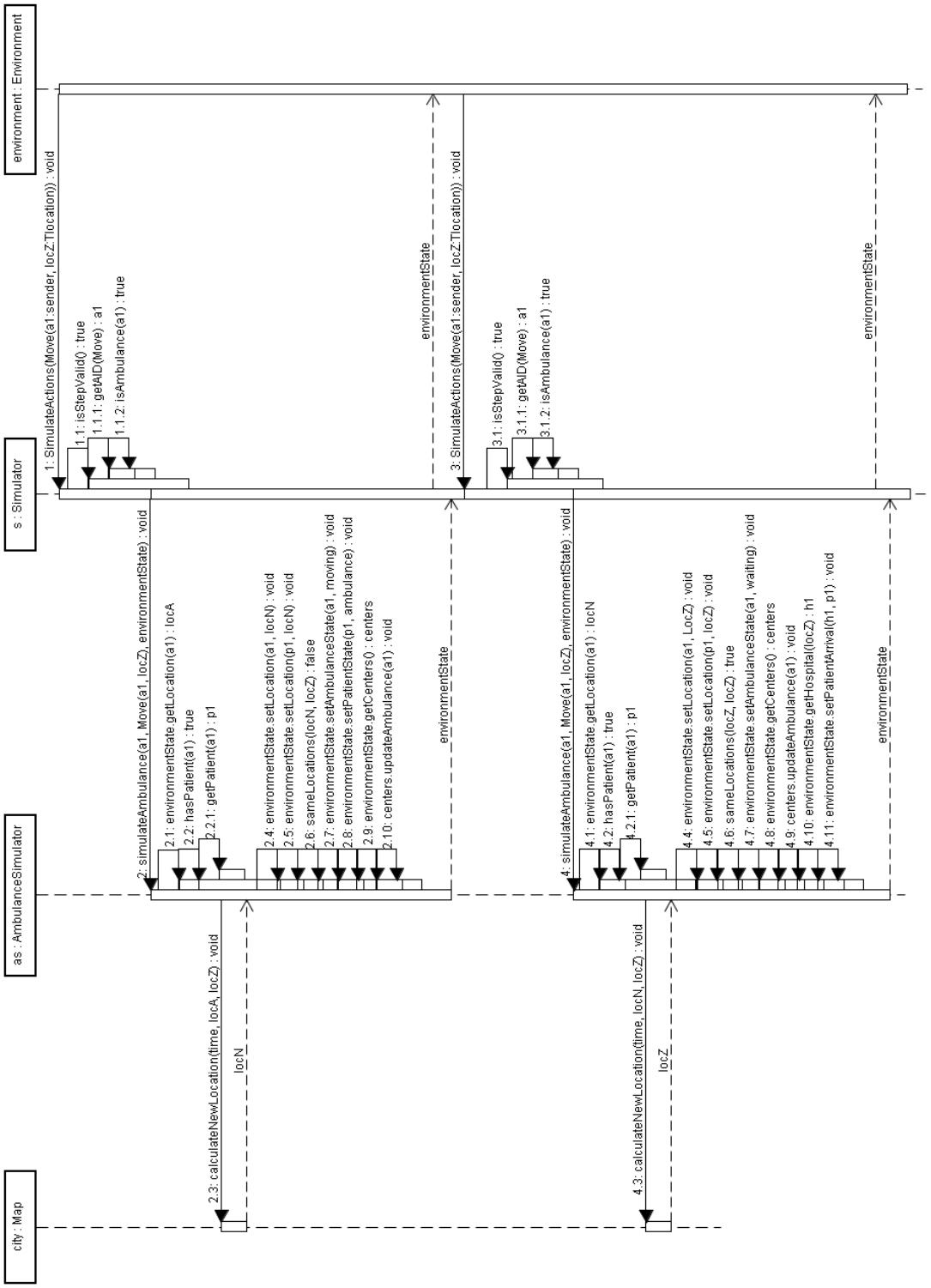


Figura 3.35: Diagrama de secuencia: La ambulancia se mueve hacia un hospital

actual y destino son las mismas. En ese caso, la ambulancia se pone en un estado de espera, *Waiting*, y se avisa al hospital que está en ese punto geográfico, que una ambulancia ha llegado a su puerta con un paciente. Igualmente, se informa a los centros de emergencias el estado de la ambulancia.

Camino Alternativo 1: la ambulancia se mueve sin estar en una misión activa. El procedimiento es análogo al anterior, con la diferencia de que sólo se modifican los estados y localizaciones de la ambulancia involucrada y al llegar a su destino, su estado pasa a ser *Stopped*, hasta que decida realizar otro movimiento. Ver diagrama B.2.7.2 del Anexo B.

- Diagramas de secuencia para simular las acciones de los hospitales

  - Simular "Admit":

    Camino Estándar: el diagrama de secuencia 3.36 muestra el proceso que se sigue para simular la admisión de un paciente en el hospital. Si el hospital decide admitir, debe comprobarse que el paciente proviene de una ambulancia. En ese caso, se elimina a la ambulancia la percepción de ese paciente, y su estado pasa de *Waiting stopped*. El paciente sale de la ambulancia y se pone en estado *Hospital*, adquiriendo la misma localización del hospital que le admite. Por último, el hospital deja de percibir ese paciente como una llegada y pasa a percibirlo como un paciente ingresado, aumentando en uno el número de camas ocupadas.

    Camino Alternativo 1: el hospital rechaza al paciente. La ambulancia cambia su estado de *Waiting* a *Stopped*, pero al seguir percibiendo en su interior al paciente, sabe que el hospital le ha desestimado su ingreso, por lo que debe continuar con el paciente y decidir si le lleva

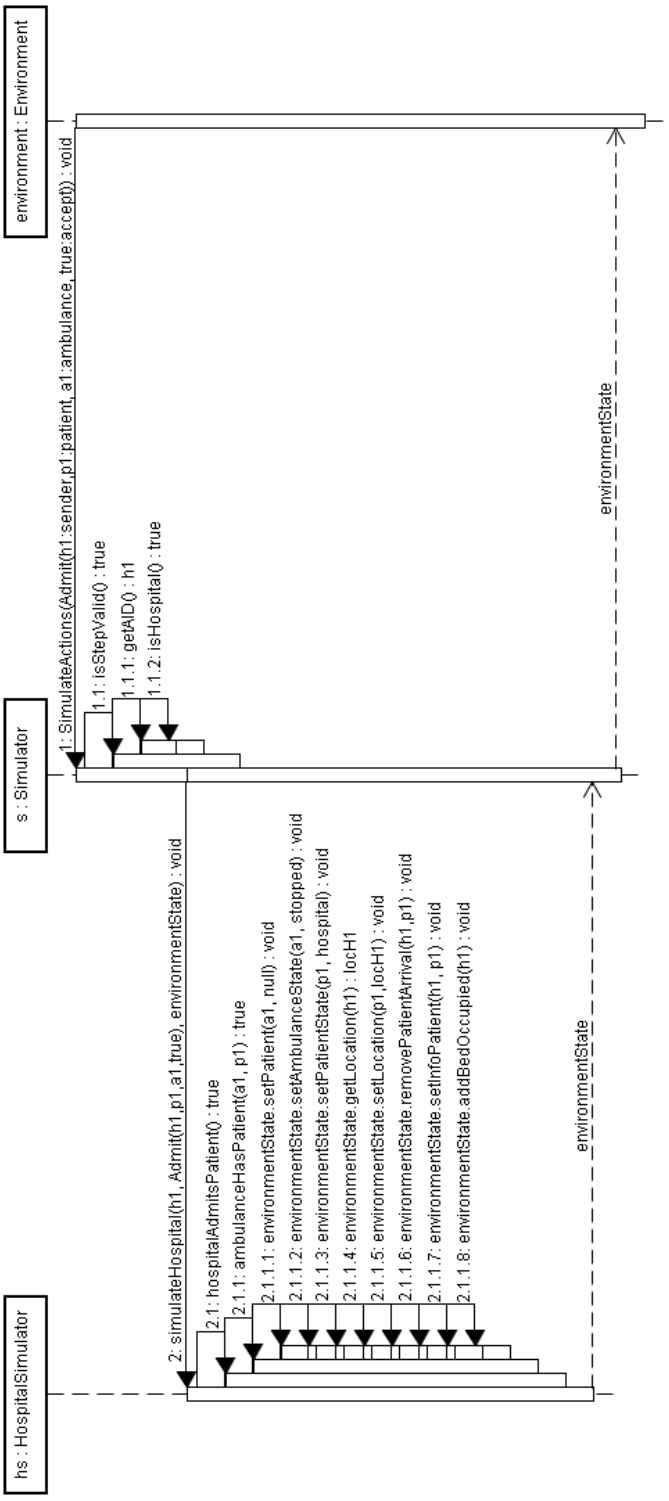


Figura 3.36: Diagrama de secuencia: Simulacion de la admisión de un paciente en el hospital

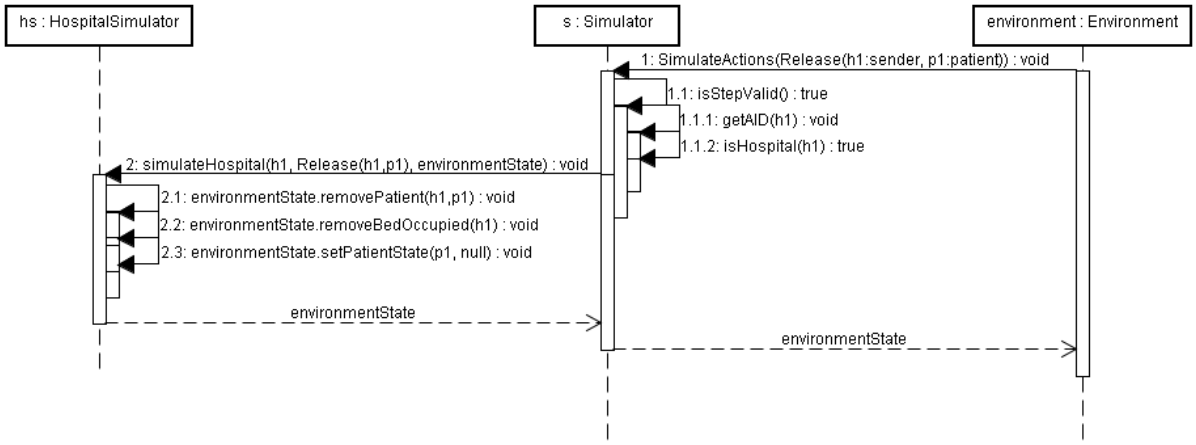


Figura 3.37: Diagrama de secuencia: Se simula el alta del paciente

a otro hospital o no. Ver diagrama B.2.9.2 del Anexo B.

- Simular "Release":

  Camino Estándar: el diagrama 3.37 explica cómo se simula la liberación de un paciente de un hospital. Cuando el hospital le da el alta, deja de percibir al paciente y deja libre una de sus camas. El paciente, pasa a un estado nulo ya que no se encuentra ni en el hospital, ni en ambulancia ni a la espera de asistencia. Si el paciente sigue enfermo tras el alta médica, deberá comenzar el proceso de llamar a un centro de emergencias y esperar asistencia. Si de lo contrario se le dio el alta por estar curado, el paciente dejará de formar parte del entorno, así que no es necesario que haga nada más.

- Diagramas de secuencia para simular las acciones de los pacientes

  - Simular "Move":

    Esta acción es trivial al movimiento de una ambulancia. Ver diagramas B.2.8.1 y B.2.8.2 del Anexo B.

- Diagramas de secuencia para procesar la vida de pacientes:

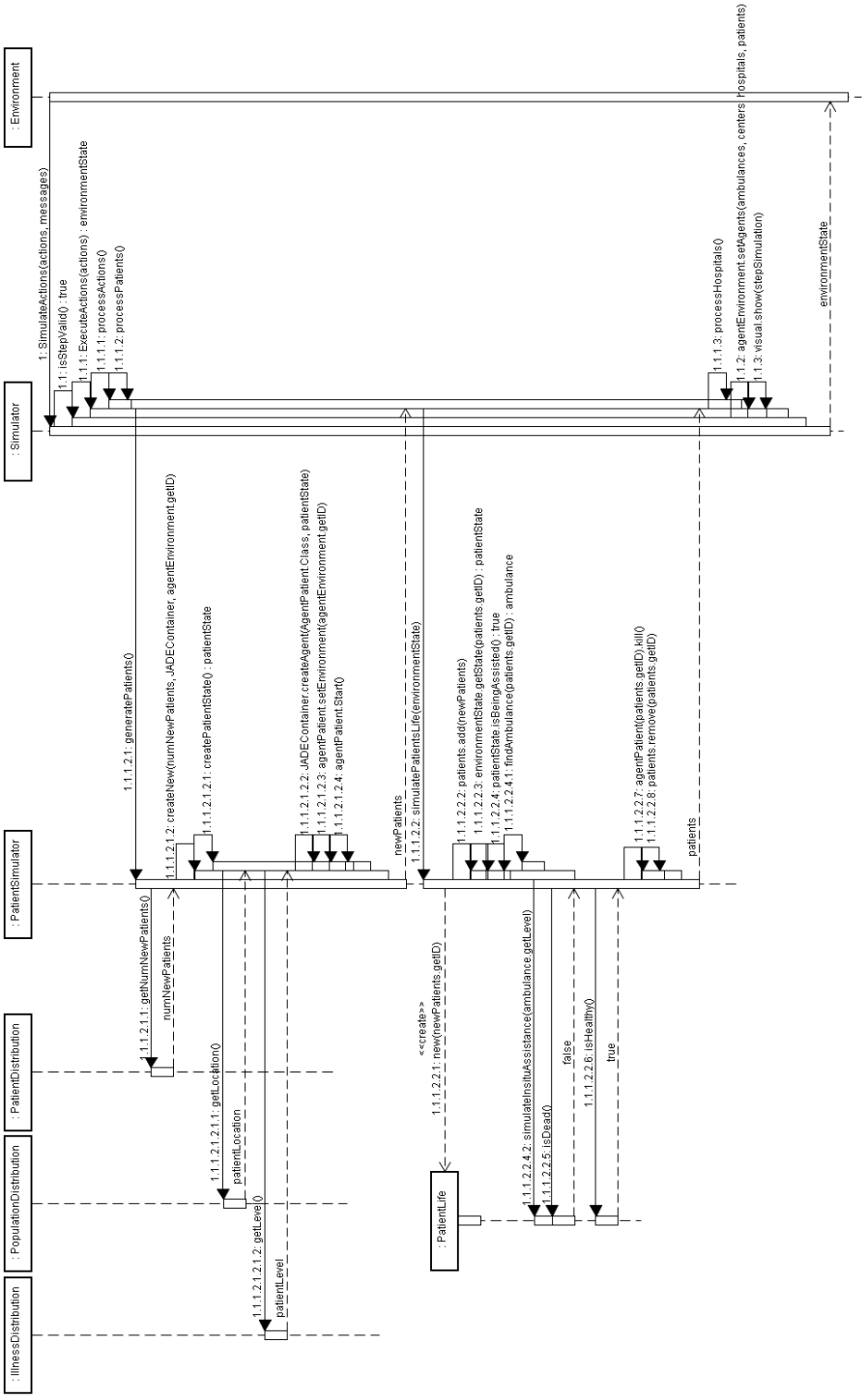


Figura 3.38: Diagrama de secuencia: simulación aleatoria de pacientes

Camino Estándar: En el diagrama de secuencia 3.38 se muestra todo el procesamiento de los pacientes. Lo primero que se hace, es generar nuevos pacientes según indique la distribución de pacientes implementada. Se crearán tantos agentes pacientes como devuelva la clase `PatientDistribution`. En la creación de cada paciente, se deben generar aleatoriamente su posición y su nivel de enfermedad. Ambos datos seguirán la distribución estadística que haya marcado el usuario en la configuración de la simulación cargada desde fichero. Por cada usuario generado, se creará un objeto `PatientLife`, encargado de almacenar los tiempos de vida y muerte que tiene ese paciente, según sea su nivel de enfermedad. Cuando ya se tienen los pacientes generados, se añaden a la lista de pacientes activos sus identificadores y a todos ellos, se les procesa su nivel de vida. En el camino básico seleccionado, se ve cómo un paciente está siendo asistido en una ambulancia. En ese caso, se busca la ambulancia que le está asistiendo y según sea su capacidad de urgencias, su nivel de vida en `PatientLife` aumentará o disminuirá con mayor o menor probabilidad. Tras ver el estado de cada paciente y calcular su tiempo de vida nuevo, se comienza a preguntar si cada uno de ellos está vivo o muerto. En este diagrama se ve cómo un paciente se ha curado tras la asistencia de una ambulancia. Cuando esto ocurre, se elimina su agente del sistema y se borra su `PatientLife` y su nombre de la lista de agentes activos para que no se vuelva a procesar en el siguiente *timeStep*. Pero su información queda almacenada en el estado del entorno, para que éste sea consciente de que el paciente se curó. Los caminos alternativos se producen en la llamada al método *`patientState.isBeingAssisted()`*, ya que cada paciente puede estar: en espera, siendo asistido por una ambulancia, siendo transportado hacia un hospital o ingresado en un hospital. Dependiendo de cada caso, se simulará su tiempo vida de una forma u otra.

Camino Alternativo 1: En el camino básico se ve cómo un paciente se cura, viendo este caso tras llamar a *`isHealthy()`* de su `PatientLife`. Un camino alternativo sería que esa llamada devolviese *false* y que el método *`isDead()`* devolviese *true*. En ese caso, se eliminaría al paciente de la lista de pacientes activos, se destruiría su agente y se borraría su `PatientLife`, para no volver a procesarlo.

Camino Alternativo 2: Si algún paciente está siendo asistido en un hospital, se simula su nivel de vida llamando al método *`simulateHospitalAssistance()`* del objeto `PatientLife` asociado a ese paciente. Para ello, se busca qué hospital es el que le tiene ingresado y, según su adecuación al nivel de enfermedad del paciente, su vida aumentará o disminuirá según proceda.

Camino Alternativo 3: Si algún paciente está siendo transportado a un hospital desde una ambulancia, se simula su nivel de vida llamando al método *`simulateAmbulanceTransport()`* del objeto `PatientLife` asociado a ese paciente. Según sea el nivel de asistencia de la ambulancia, su vida se estabilizará o disminuirá con mayor o menor probabilidad.

Camino Alternativo 4: Si algún paciente está esperando asistencia médica, siempre se decrementa su nivel de vida con el método *`decrementLife()`* de su `PatientLife` asociado.

- Diagramas de secuencia para procesar la ocupación de los hospitales:

  Camino Estándar: Un requisito de la aplicación era incorporar aleatoriedad a la ocupación de las camas de los hospitales, ya que de esa forma se simula que no solo llegan a los hospitales pacientes desde las ambulancias, sino que también pueden llegar por su propio pié, trasladados desde otros hospitales etc. Por ello, en el diagrama de secuencia 3.39 se ve como después

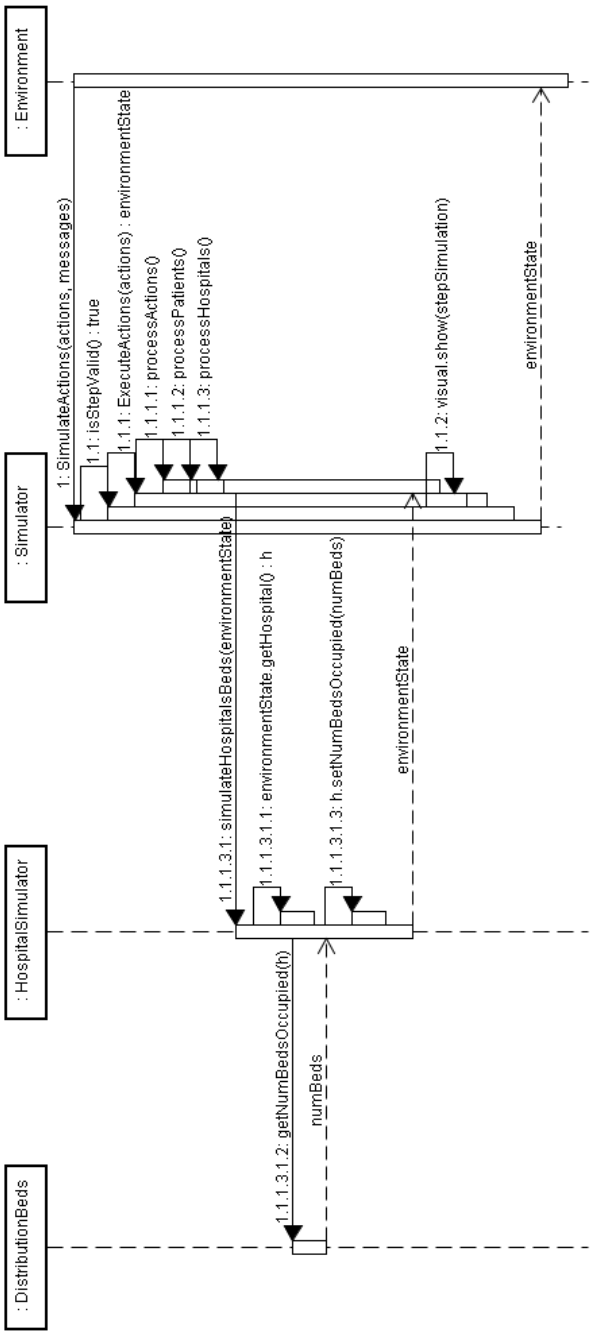


Figura 3.39: Diagrama de secuencia: simulación aleatoria de hospitales

de simular todas las acciones, se procesa la ocupación de las camas de cada hospital. Se toma el estado interno de cada hospital que existe en el entorno y se solicita a la clase que distribuye el número de camas, que devuelva la ocupación que debe tener en ese momento el hospital seleccionado. A su estado interno, se le modifica la ocupación y se actualiza con el valor devuelto por `DistributionBeds`. Este objeto tendrá implementado el tipo distribución elegida por el usuario en la configuración inicial cargada, y devolverá un valor del número de camas ocupadas según sea esa distribución.

3.5.2.2.4. Salvar paso de simulación

Camino Estándar: En el diagrama de secuencia 3.40, se detalla el proceso para guardar cada paso de simulación. Una vez se han simulado las acciones recibidas, se envía al objeto `Historical` la lista de acciones recibidas, los mensajes que se han enviado entre los agentes y el estado del entorno que ha quedado tras la simulación de dichas acciones. Con toda esa información, se crea una instancia de la clase `StepSimulation` y dicho objeto se almacena en el fichero histórico. Tras el almacenamiento, la clase `Simulator` pedirá a `Historical` el último paso de simulación guardado y se lo enviará al módulo *Display* para que lo muestre, si el usuario seleccionó la visualización.

3.5.2.2.5. Finalizar la simulación

Camino Estándar: Cuando el simulador recibe una lista de acciones por parte del entorno, éste comprueba que sea un paso de simulación válido, es decir, que no se haya superado el número de pasos máximos definido en la configuración inicial de la simulación. Si se llega al *step* máximo, se da la orden de finalizar la ejecución de la simulación, tal y como se ve en el diagrama 3.41. Esto conlleva cerrar el fichero histórico y destruir a todos los agentes que estaban creados en el

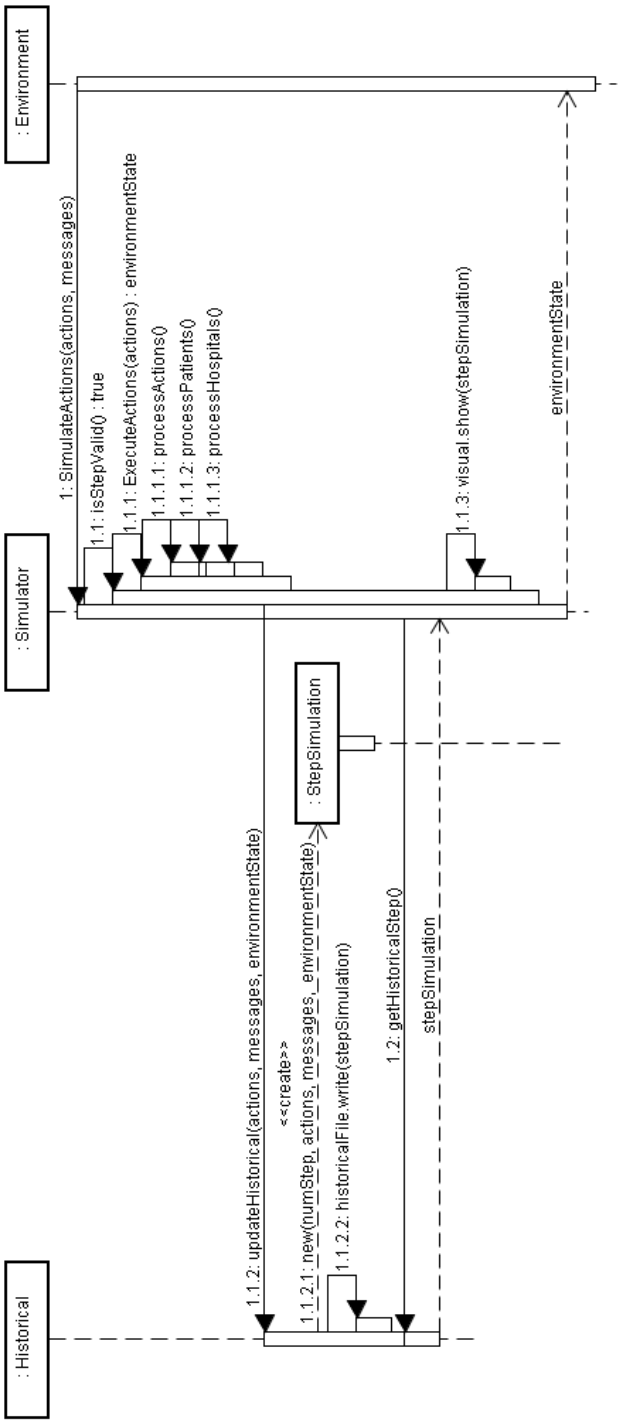


Figura 3.40: Diagrama de secuencia: guardar los pasos de simulación

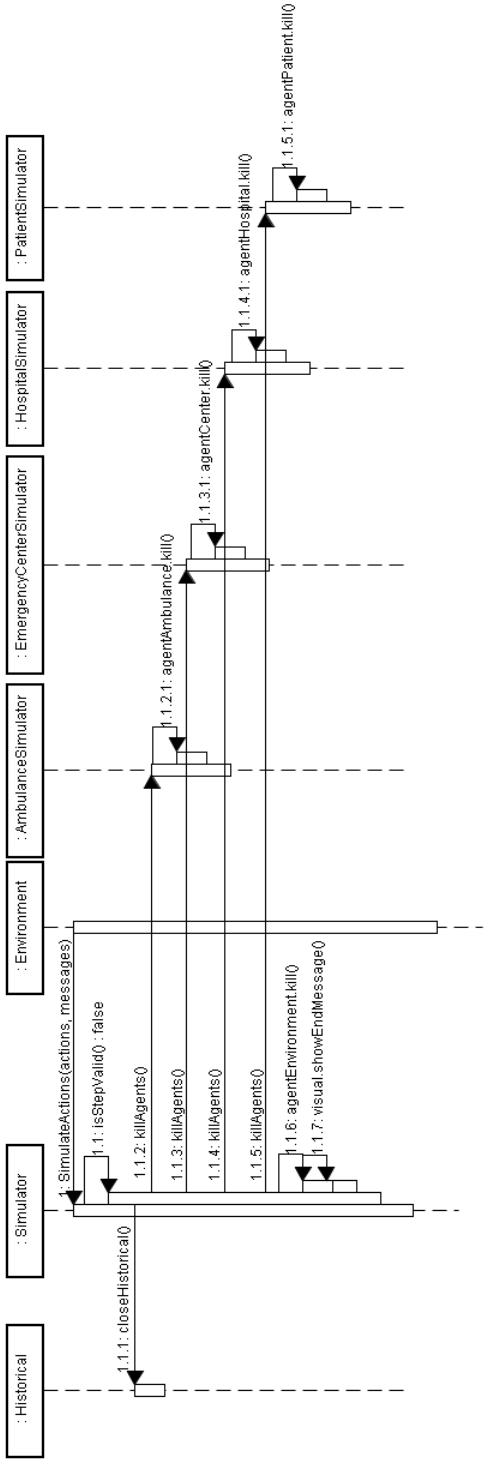


Figura 3.41: Diagrama de secuencia: finalización de la simulación

sistema, incluido el entorno, para finalizar la comunicación entre todos ellos. Cada simulador de agentes sera el encargado de eliminar sus agentes correspondientes, ya que esas clases son las que almacenan las listas de agentes vivos en el entorno. Después, se mostrará al usuario un mensaje para indicarle que la simulación se ha finalizado y él podrá volver a la pantalla de inicio a comenzar otra simulación, crear una nueva configuración o visualizar una simulación ya existente.

3.5.2.2.6. Crear configuración inicial y salvarla en un fichero

Camino Estándar: al comienzo de ejecutar una simulación, se debe cargar un fichero de configuración con sus parámetros. Esta configuración se crea con el módulo de *Simulador* llamado *Setup*, tal y como se muestra en el diagrama de secuencia 3.42. Con él, el usuario puede definir todos los parámetros que caracterizarán a la simulación, pasando por una serie de pantallas que le van solicitando datos acerca del escenario de urgencias que se vaya a configurar. Al finalizar, se guarda esa información en un fichero para su posterior uso, denominado fichero de configuración.

Camino Alternativo 1: cada vez que el usuario desea pasar de una pantalla a otra para seguir introduciendo datos, el sistema comprueba que el usuario haya completado toda la información requerida en ese paso. Si algún campo está incompleto, se le muestra al usuario y mensaje de error y se le vuelve a pedir el dato faltante, de forma que pueda seguir completando el proceso de configuración.

Camino Alternativo 2: al final de la configuración, los datos introducidos por el usuario se almacenan en un fichero para su posterior uso. Si existe algún problema en la escritura de ese fichero, ya sea por no tener permisos de escritura en el directorio especificado o por haber dado un nombre no válido de simulación, el sistema informará del error y se volverá a la pantalla de inicio para repetir, si el usuario desea, el proceso de configuración.

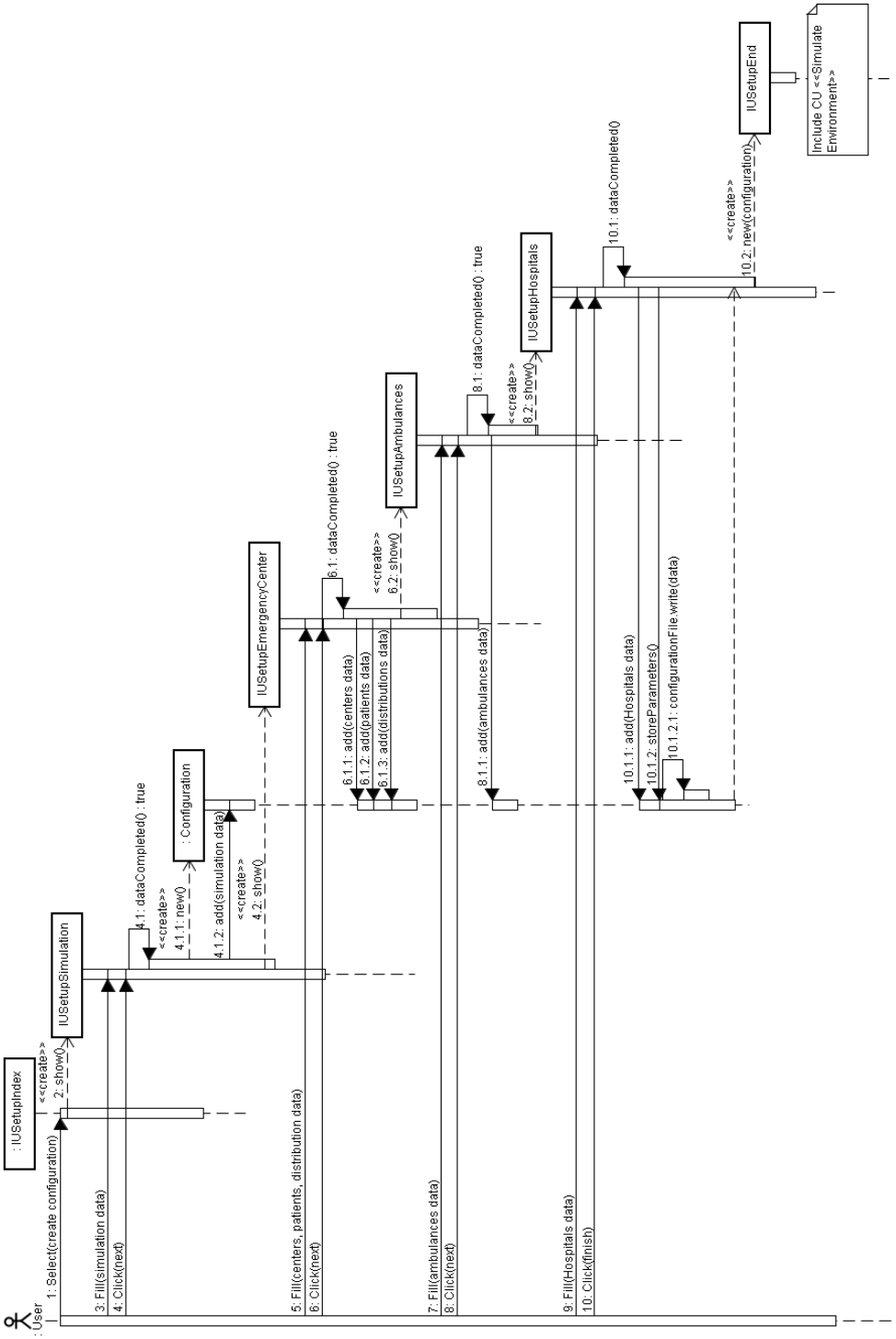


Figura 3.42: Diagrama de secuencia: configurar los parámetros de una simulación

3.5.2.3. Visualizador

Las clases que componen el subsistema *Visualizador* se encargan de mostrar al usuario la información generada en la realización de una simulación. Esa información se podrá ver con mayor o menor nivel de detalle, seleccionando cada agente de forma individual o viendo la información de la simulación de forma global, respectivamente. Las secuencias de acciones que hacen posibles estas funciones se recogen en los siguientes apartados:

3.5.2.3.1. Visualizar la simulación de forma global

Camino Estándar: cada vez que el entorno manda simular una lista de acciones al simulador, éste guarda la información generada en un histórico y actualiza la interfaz de usuario del módulo *Visualizador* cuando el usuario selecciona la opción de visualización al ejecutar el programa, tal y como se puede apreciar en el diagrama 3.43. La información que el subsistema *Simulador* envía al *Visualizador* se procesa a través de una clase llamada `AgentsInformation`, la cual recibe el último paso de simulación del histórico y transforma esa clase a una información legible por el usuario. Además, ese objeto `StepSimulation` lo transforma a formato JSON para que la página Web encargada de mostrar el mapa y los datos de simulación sobre él, pueda manejar esa información correctamente. Después de esto, se llama a la clase `IUSimulation` y se le pide que se actualice mostrando la nueva información del `AgentsInformation` en sus componentes gráficos: un árbol con información resumida de cada agente, clasificados por tipo de actor, un navegador Web con el mapa del entorno y una barra de progreso que indica la evolución de la simulación.

Camino Alternativo 1: si el simulador pide a la clase `IUSimulation` que dibuje un paso de simulación cuyo número de orden es igual o superior al tamaño

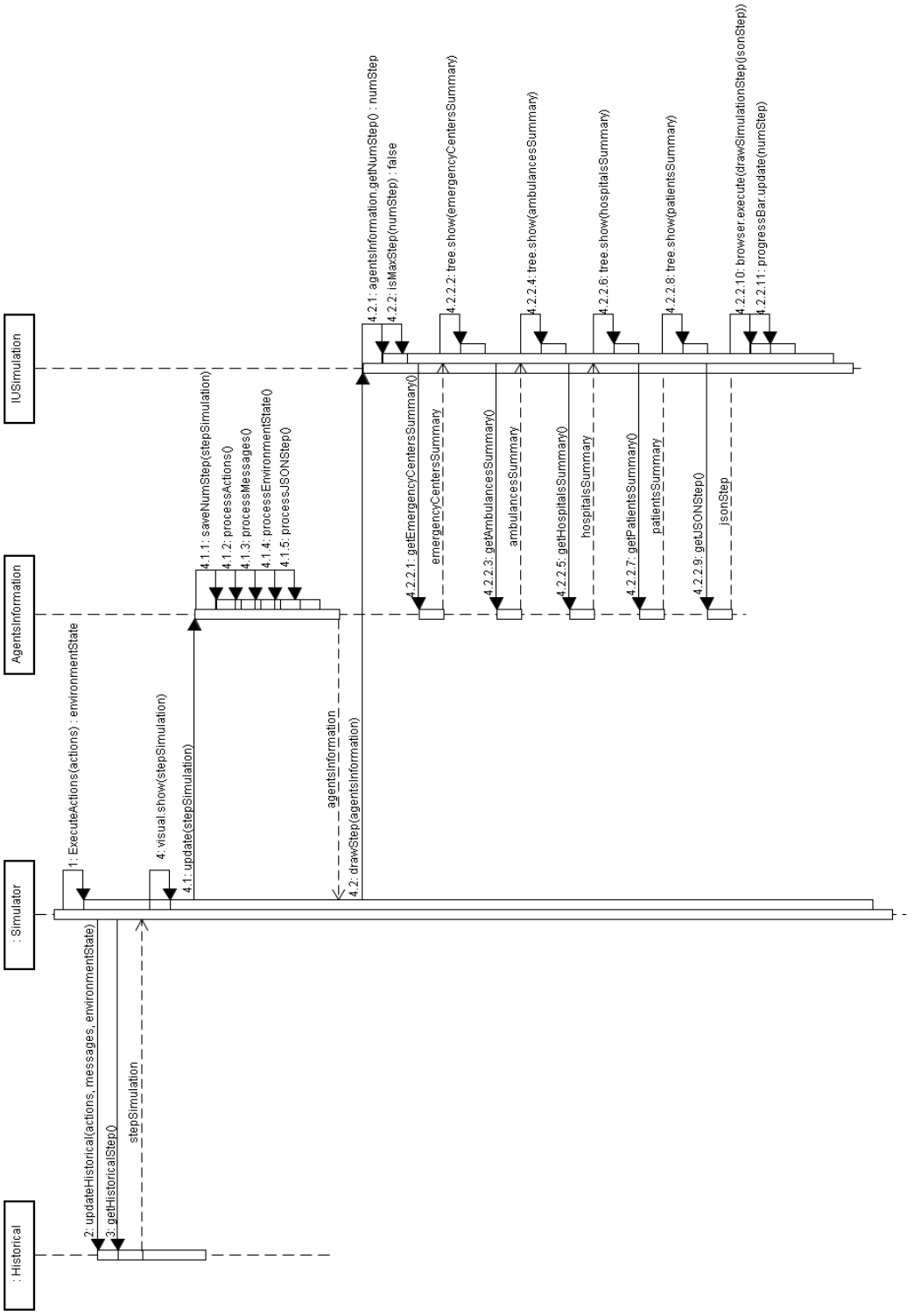


Figura 3.43: Diagrama de secuencia: visualizar una simulación

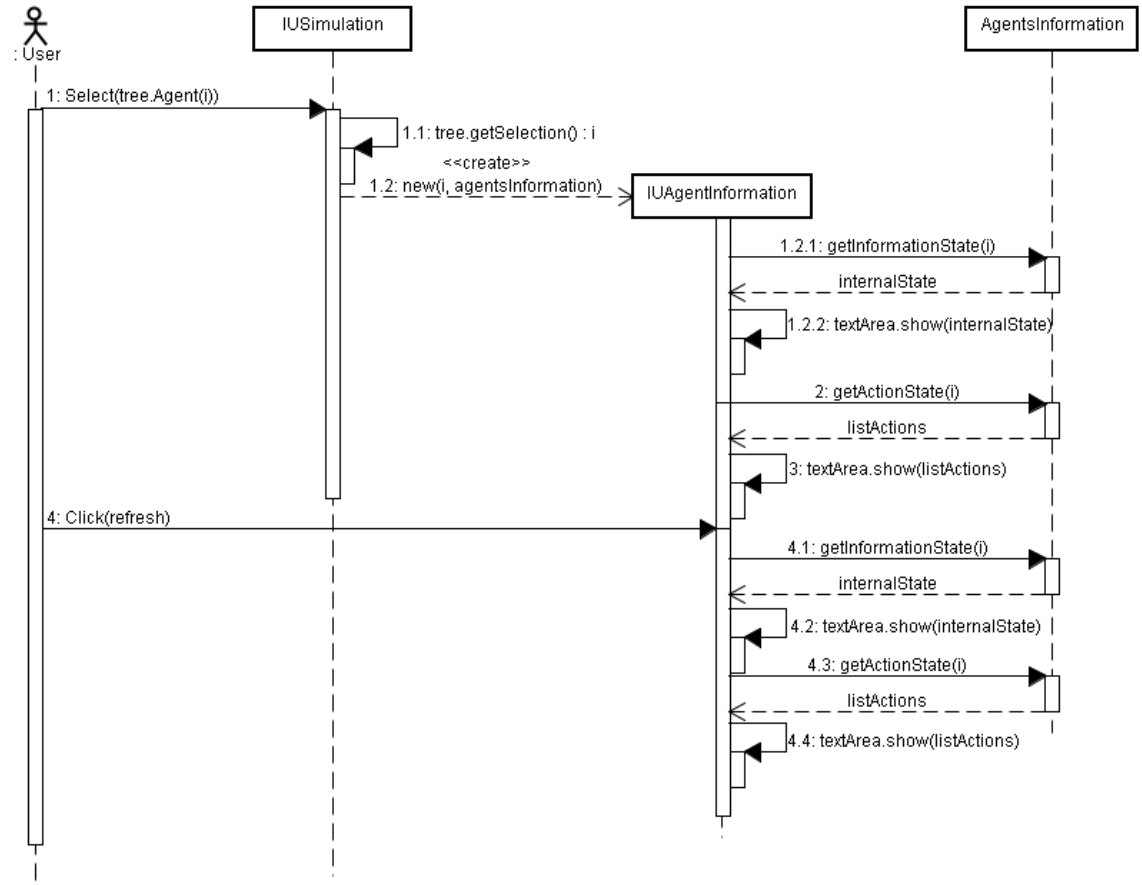


Figura 3.44: Diagrama de secuencia: visualizar la información de cada agente

máximo de pasos de simulación indicados por el usuario en la configuración inicial, el sistema debe mostrar un mensaje informativo indicando que la simulación ha llegado a su fin, completando la barra de progreso y dejando el resto de componentes con la información del último paso de simulación realizado. Tras ese mensaje, el usuario podrá cerrar la ventana `IUSimulation` y volver al inicio de la aplicación.

3.5.2.3.2. Visualizar la información asociada a un único agente

Camino Estándar: en el diagrama 3.44 se muestra la secuencia de acciones para mostrar los datos de un agente del entorno de forma individual. Cuando un usuario selecciona uno de los nodos del árbol de la ventana `IUSimulation`, se le muestra una ventana modal con la información del agente asociado a ese nodo. Esa información incluye el estado interno actual de ese agente y el histórico de acciones que ha ido realizando en lo que va de simulación. Si el usuario presiona el botón "*Refresh*", esta información se actualizará, solicitando al objeto `AgentsInformation` los nuevos datos.

3.5.3. Diseño de la interfaz de usuario

El usuario va a interactuar con el sistema mediante una serie de interfaces gráficas a lo largo de la ejecución del programa, las cuales servirán tanto para mostrarle información como para solicitarle a él datos que el sistema necesite.

Las clases visuales del proyecto son aquellas que comienzan con las siglas "IU" (Interfaz de Usuario), recogidas en el diagrama de clases 3.45. Por un lado, hay un grupo de clases implicadas en la creación de la configuración de la simulación (*Simulation Setup*) y otras clases forman parte de la visualización final del entorno simulado (*Visualizador*). Como se puede ver en el diagrama, todas las clases visuales tienen un componente gráfico de la librería SWT (librería para crear interfaces gráficas en Java) llamado *Shell*. Este componente representa una ventana y se encarga de administrar todo lo que tenga en su interior: botones, texto, imágenes, etc. Cada clase visual se va a corresponder con una ventana diferente, por lo que cada una de ellas necesita tener una *Shell* como componente principal.

A continuación, se explica con más detalle el contenido de cada una de estas

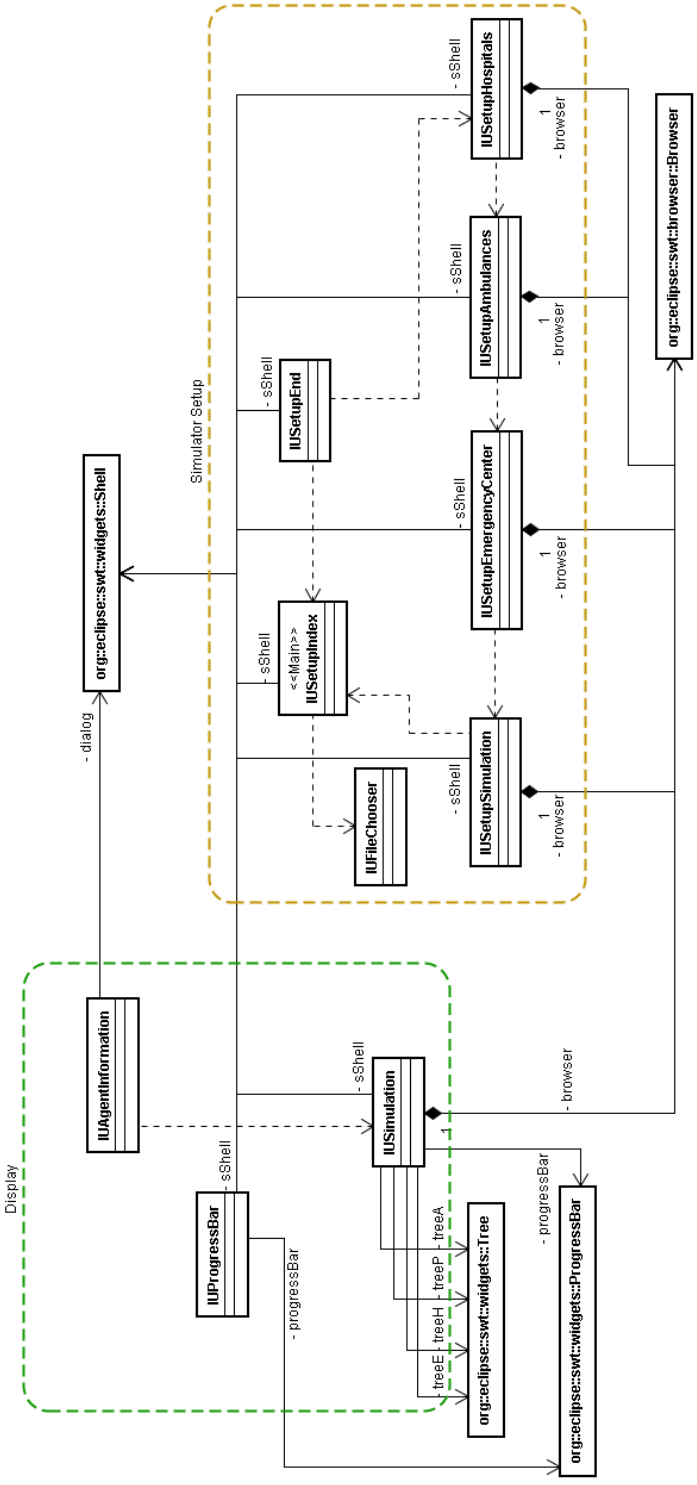


Figura 3.45: Diagrama de clases de la interfaz de usuario

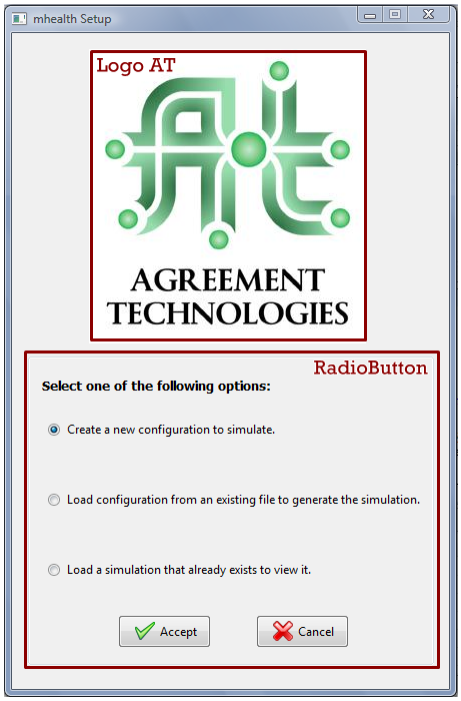


Figura 3.46: Ventana IUSetupIndex

ventanas, su utilidad y la relación con las demás clases visuales:

3.5.3.1. Setup

Las clases que recogen esta funcionalidad se van a encargar de solicitar al usuario la información necesaria para poder ejecutar una simulación. Esa información consiste en todos los parámetros configurables de la misma, así como las ubicaciones de los ficheros que se generarán.

- IUSetupIndex: esta es la pantalla principal de todo el proyecto y la que contiene el método *Main* que lanza el programa.

  Se compone de tres `RadioButtons` para que el usuario elija una de las tres funcionalidades generales del sistema:

  1. Crear la configuración de una simulación: esta opción llevará al usuario

a la ventana *IUSetupSimulation*.

2. Ejecutar una simulación ya configurada: se solicitará el fichero de configuración mediante *IUFileChooser* y después se irá a *IUSetupEnd*.
3. Visualizar una simulación previamente realizada: se solicitará el fichero con el histórico de la simulación mediante *IUFileChooser* y la clase *Player* (ver diagrama 3.23) irá leyendo el contenido para mostrarlo en la ventana de *IUSimulation*.

- IUFileChooser: esta clase se utiliza en aquellos casos en los que se quiere guardar un fichero o en los que se quiere cargarlo. Abrirá una ventana para seleccionar la ruta de un fichero existente en el ordenador cuando se quiere cargar un fichero de configuración o de simulación. Cuando se quiere salvar un archivo, se abrirá una ventana para elegir la ubicación de este en la máquina donde está corriendo el programa. Este último caso ocurrirá al crear configuraciones, en la ventana de la clase *IUSetupSimulation*, ya que el usuario puede elegir en esta clase dónde quiere que se almacenen los ficheros que se crearán de configuración y el histórico asociado.

- IUSetupSimulation: la funcionalidad de esta clase consiste en pedir al usuario datos sobre la simulación en general.

  La ventana de esta clase se divide en tres partes bien diferenciadas. El objetivo de la primera es pedir al usuario el identificador de la configuración que va a crear y las ubicaciones de los ficheros que se generarán sobre la simulación (se usará la clase *IUFileChooser* al pinchar en los botones de búsqueda de directorio).

  La segunda área maneja los tiempos de la simulación: lo que durará cada *timeStep*, los *timeSteps* que tendrá en total la simulación y la escala de tiempo que se usará para simular los movimientos.

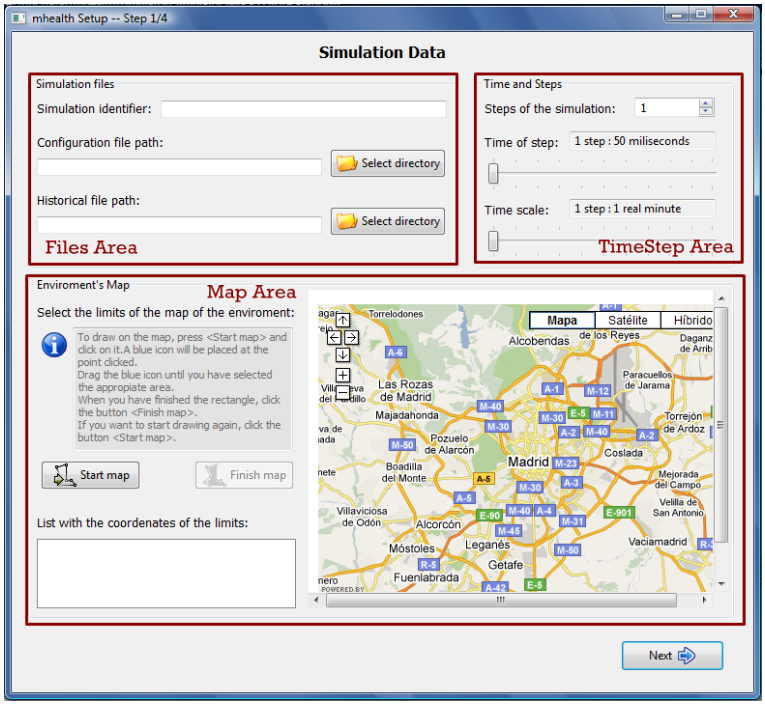


Figura 3.47: Ventana IUSetupSimulation

La parte inferior de la ventana se compone de un navegador (clase *Browser* de la librería SWT) que cargará la página Web que muestra un mapa del mundo. Servirá para seleccionar un área del mapa, de forma que el escenario de emergencias médicas que se simulará esté comprendido entre los limites seleccionados. Una vez se completen todos los datos de este formulario, se mostrará al usuario la pantalla *IUSetupEmergencyCenter*.

- IUSetupEmergencyCenter: esta clase se encarga de solicitar al usuario los datos referentes a los centros de emergencia que existirán en el escenario configurado, los datos sobre los pacientes del entorno y las distribuciones estadísticas que dotarán al simulador de aleatoriedad en algunos aspectos.

  El formulario sobre los centros de coordinación se compone de un área de texto para introducir el identificador del centro, un componente para

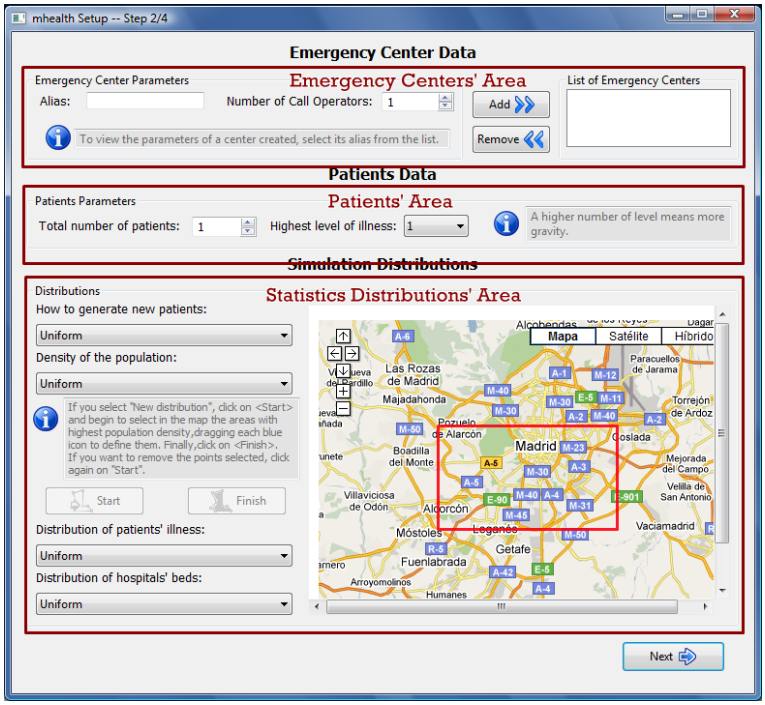


Figura 3.48: Ventana IUSetupEmergencyCenter

seleccionar el número de operadores telefónicos que formarán la centralita del centro de emergencias y el listado de los centros que se han ido añadiendo al entorno.

El formulario sobre los pacientes.

La zona de las distribuciones estadísticas.

- IUSetupAmbulances: esta clase pide al usuario todos los datos necesarios para añadir ambulancias a nuestro entorno. Existen tres áreas en esta ventana: una sección para introducir los diferentes parámetros que definirán una ambulancia, como su alias, capacidad de atención sanitaria según los niveles de enfermedad de los pacientes y la ruta que siguen en el mapa del escenario definido anteriormente; un listado de las ambulancias que se han ido creando en la parte izquierda de la ventana y por último, una zona para

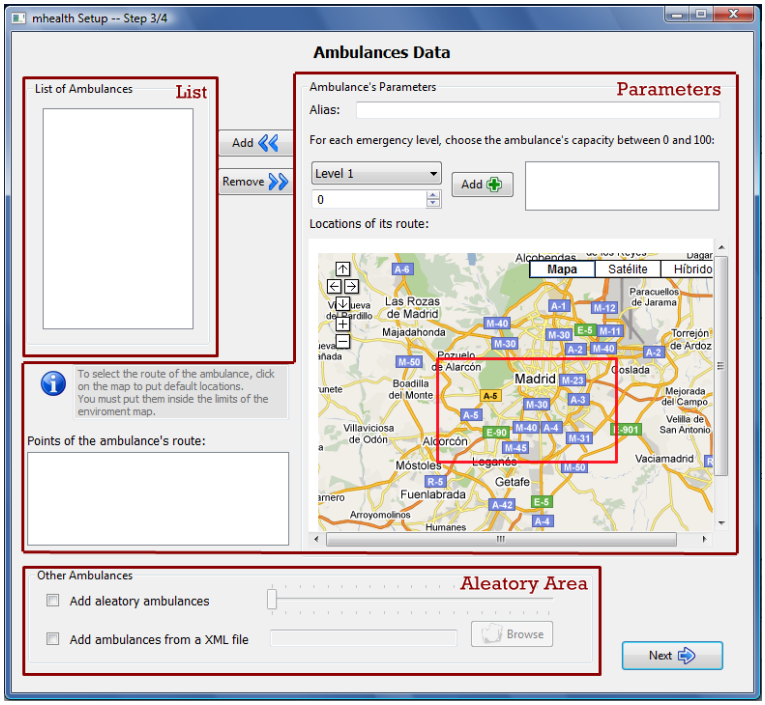


Figura 3.49: Ventana IUSetupAmbulances

agregar ambulancias sin especificar una a una sus parámetros ya sea de forma aleatoria o cargando un fichero que almacena datos de ambulancias.

Una vez se completen todos los datos de este formulario, se mostrará al usuario la pantalla *IUSetupHospitals*.

- IUSetupHospitals: esta clase se encarga de solicitar al usuario todos los datos necesarios para añadir hospitales a nuestro escenario de emergencias médicas. Existen tres áreas en esta ventana: una sección para introducir los diferentes parámetros que definirán un hospital, como su alias, el número de miembros que hay en urgencias, número de camas, nivel máximo de gravedad de pacientes que es capaz de atender en urgencias y su localización; un listado de los hospitales que se han ido creando en la parte izquierda de la ventana y por último, en la parte inferior, una zona para agregar

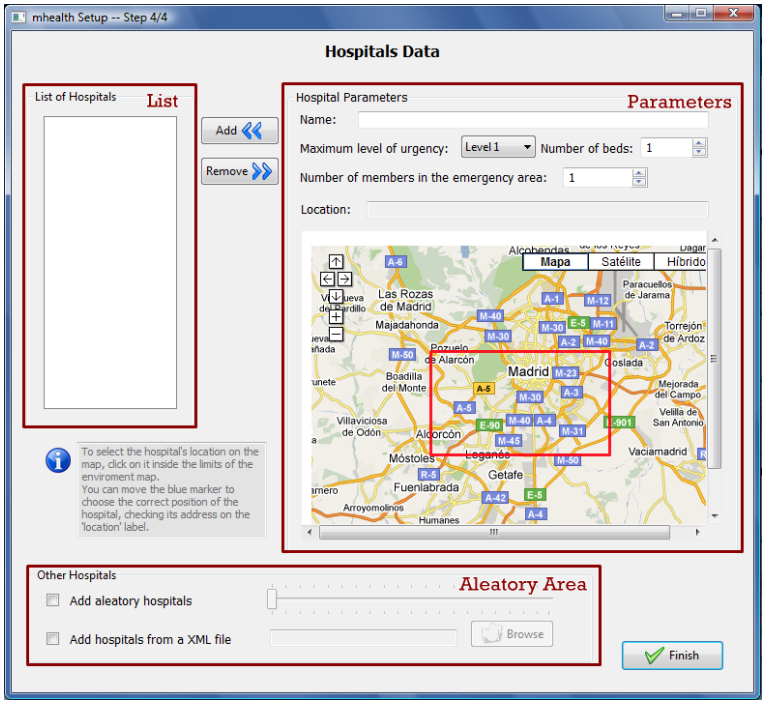


Figura 3.50: Ventana IUSetupHospitals

hospitales sin especificar uno a uno sus parámetros ya sea de forma aleatoria o cargando un fichero que almacena datos de hospitales.

Una vez se completen todos los datos de este formulario, el usuario habrá completado todo el proceso de configuración y se le llevará a la pantalla *IUSetupEnd*.

- IUSetupEnd: ventana final del proceso de configuración. También aparece cuando se elige la segunda opción de "carga de un fichero de configuración" para comenzar una simulación. La zona superior de la ventana indica al usuario los directorios donde se encuentran los ficheros de configuración e histórico tal y como él lo definió en la pantalla *IUSetupSimulation*.

  En la zona inferior se ofrecen 2 opciones de visualización de la simulación: ver la ejecución en tiempo real gráficamente, mediante la pantalla *IUSimu-*

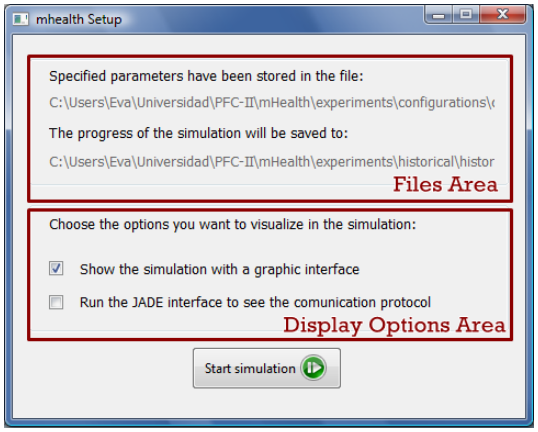


Figura 3.51: Ventana IUSetupEnd

lation (subsistema *Visualizador*) o lanzar la interfaz gráfica de la plataforma JADE para ver cómo se va llevando a cabo la comunicación entre los agentes. Si la primera opción no es seleccionada, se mostrará la ventana *IUProgressBar* para que el usuario vea el progreso de la simulación elegida y se le avise cuando ha finalizado, para que pueda volver a *IUSetupIndex* y poder cargar el histórico de esas simulación o proceder con la ejecución de otras simulaciones.

3.5.3.2. Visualizador

Las clases que componen este subsistema se van a encargar de mostrar al usuario la información generada en la realización de una simulación. Esa información se podrá ver con mayor o menor nivel de detalle y se estructura de la siguiente forma:

- IUSimulation: ventana principal de la visualización. Es la encargada de recoger todos los agentes que están actuando en el entorno y mostrar la información básica de cada uno para tener una visión global de lo que está pasando en cada momento en el escenario de la simulación. La ven-

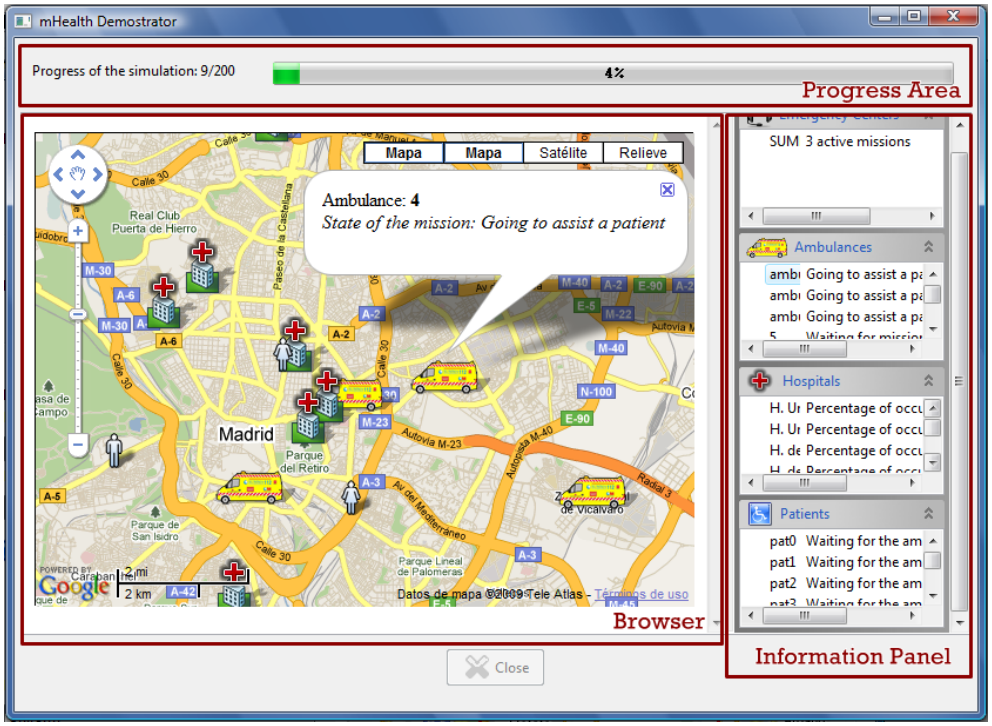


Figura 3.52: Ventana IUSimulation

tana se subdivide en tres zonas, cada una con un objetivo diferente. La zona superior contiene una barra de progreso e información en formato textual de la duración de la simulación y el transcurso de ésta, para que el usuario conozca en cada momento el tiempo que dura la simulación que se está llevando a cabo.

La segunda zona, la situada a la izquierda de la pantalla, contiene un navegador Web que muestra el mapa del entorno. En él se pueden ver iconos representativos de cada agente activo en el entorno y los movimientos que van realizando.

La tercera zona, a la derecha, se compone de un panel de información con datos de los agentes clasificados por categoria. Cada categoría es un área que se puede expander o contraer, y en su interior se numeran los agentes

con un texto resumen de su situación actual. Cuando el usuario pinche sobre el identificador de cada agente en este panel, se le abrirá una ventana *IUAgentInformation* con mayor nivel de detalle sobre el estado de cada agente.

Finalmente, cuando la simulación se haya completado, el botón inferior permitirá al usuario volver a *IUSetupIndex* para que pueda continuar usando el programa.

- IUAgentInformation: esta clase se corresponde con un diálogo modal de la clase *IUSimulation*. Esto quiere decir que *IUSimulation* es la ventana padre de ésta, y no se cerrará cuando se abra una ventana *IUAgentInformation*. De esta forma, se pueden abrir tantas ventanas de información de los agentes como se deseen, y así poder ir viendo los datos completos de diferentes agentes a la vez. Esta ventana está formada por dos áreas de información: la superior, que muestra el estado del agente y la inferior, un histórico de acciones.

  En el estado del agente se muestran todos los parámetros que dan forma al agente en el momento de abrir la ventana. Cada tipo de agente tiene sus propios parámetros, por lo que este área de texto será diferente según el agente seleccionado en *IUSimulation*.

  Abajo, se muestra un área de texto con el histórico de acciones que lleva el agente realizadas hasta ese momento, indicando en qué *timeStep* se hizo cada una de ellas.

  Finalmente, se tiene un botón para poder actualizar la información de toda la ventana y ver cómo evoluciona el estado del agente y su histórico de acciones simuladas.

- IUProgressBar: el contenido de esta clase visual se muestra cuando el usuario

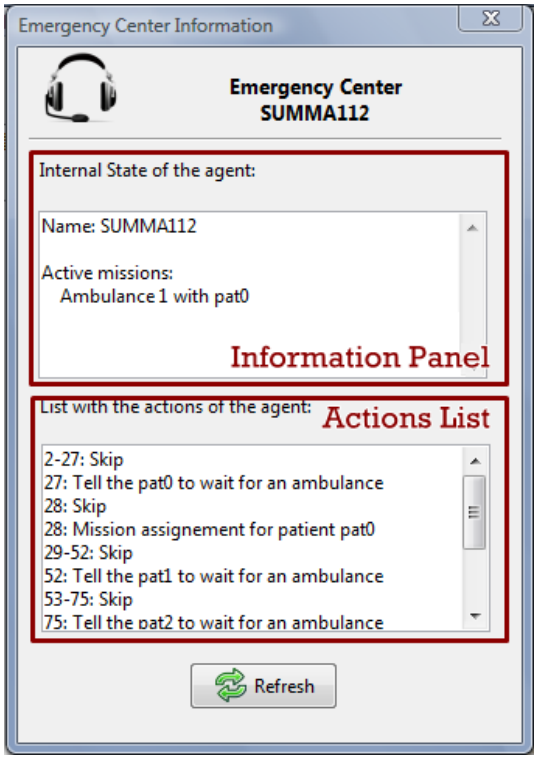


Figura 3.53: Ventana IUAgentInformation

quiere realizar una simulación sin visualizarla en ese momento. En ella, el

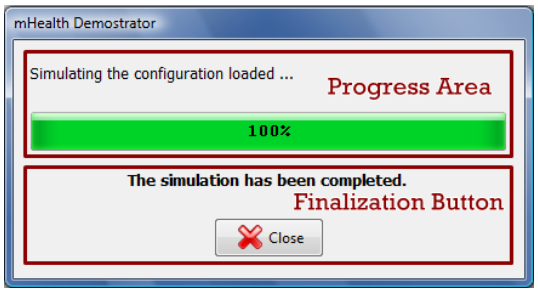


Figura 3.54: Ventana IUProgressBar

usuario puede ver el progreso de la simulación elegida mediante una barra de progreso. Cuando termine la ejecución, la barra de progreso se habrá completado y el usuario podrá volver a *IUSetupIndex* mediante el botón inferior de la ventana.

3.6. Implementación

En la fase de implementación, se distribuye el sistema en términos de componentes, que son empaquetamientos físicos de uno o varios elementos del modelo de diseño. A través de unos diagramas de despliegue, se va a especificar cómo se organizan esos componentes y cómo dependen unos de otros dentro del sistema implementado. Esa organización de los componentes, junto con los detalles más importantes de implementación llevados a cabo y las herramientas utilizadas para desarrollar el proyecto, son explicados en esta sección de la memoria.

3.6.1. Diagramas de despliegue

Para dar una vista física de todo el proyecto presentamos a continuación un diagrama de despliegue, ver 3.55, cuya finalidad es mostrar los nodos individuales del sistema y sus enlaces y componentes que ejecutan en cada uno. En la figura 3.55, podemos distinguir dos nodos fundamentales, el nodo en el que ejecuta la *Aplicación* (este nodo puede estar distribuido, ver 3.6.1.1), y el nodo en el que ejecuta el *Simulador* y el *Visualizador*.

3.6.1.1. Aplicación

El subsistema *Aplicación* depende del subsistema *Simulador*, ya que los contenedores de los distintos agentes pertenecen al contenedor creado en el *Simulador*. Cabe destacar que que la *Aplicación* es totalmente distribuida, es decir, en el diagrama 3.56 vemos que se ha creado un contenedor distinto para cada tipo de agente en maquinas diferentes, esta configuración es variable, ya que todos los agentes pueden estar en un mismo nodo, e incluso utilizar un nodo para cada agente.

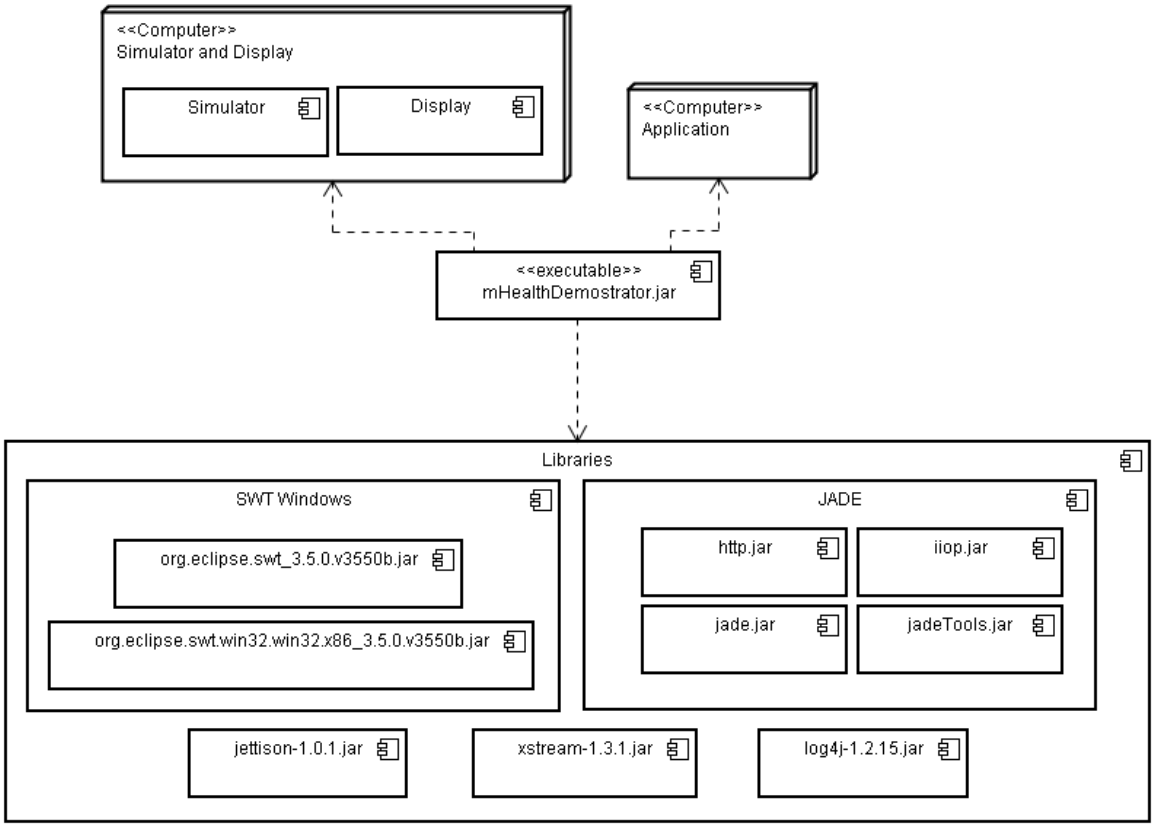


Figura 3.55: Diagrama de despliegue de los componentes

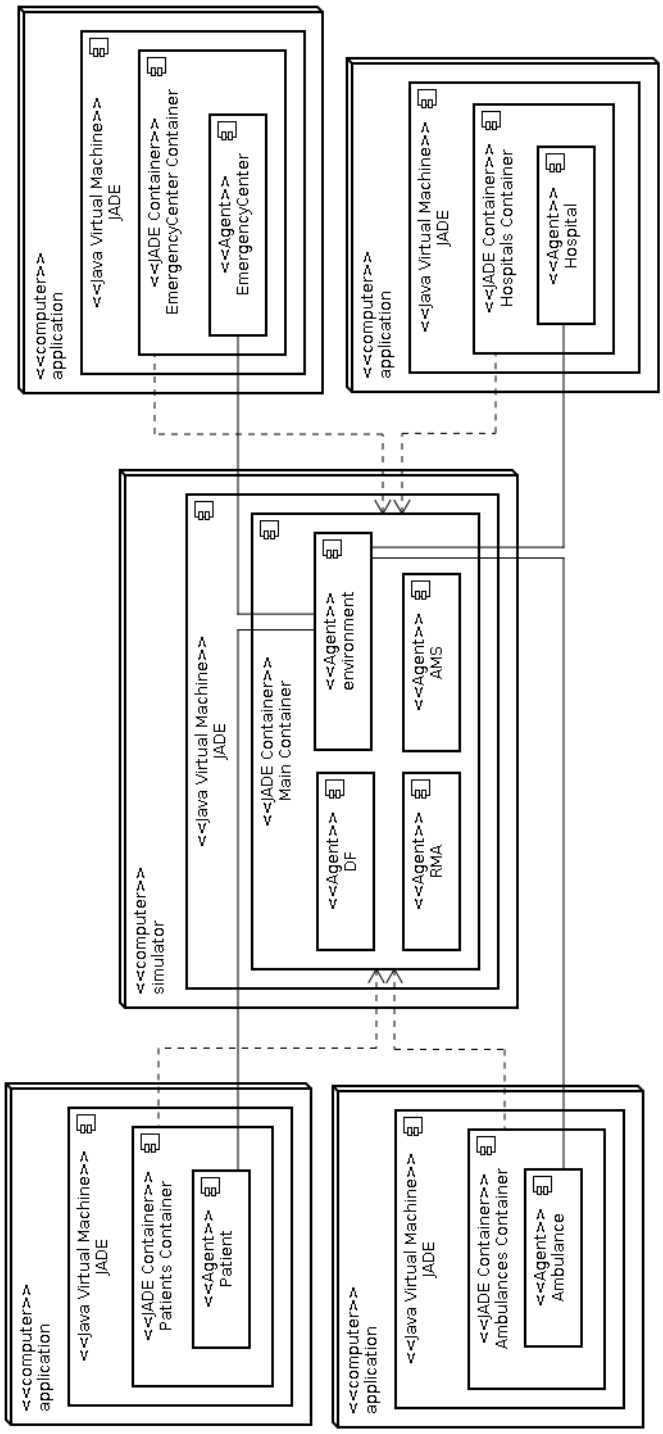


Figura 3.56: Diagrama de despliegue del subsistema *Aplicación*

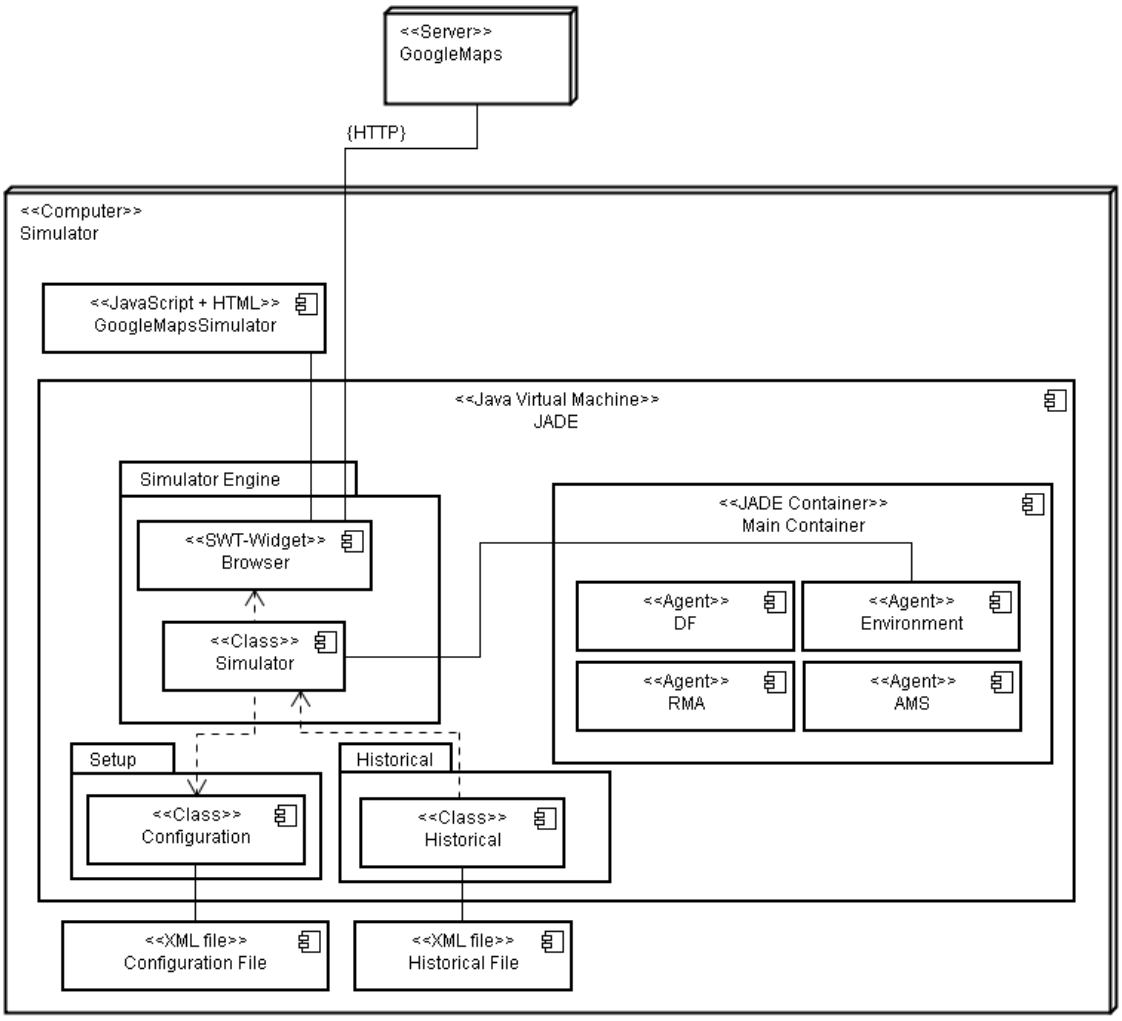


Figura 3.57: Diagrama de despliegue del subsistema *Simulador*

3.6.1.2. Simulador

El subsistema *Simulador*, como podemos ver en la figura 3.57, contiene código java ejecutándose, en el cual hay dos partes diferenciadas: la primera de ellas es el contenedor en el que se crea en el entorno y en el cual se añadirán los agentes de la *Aplicación* y la segunda es el motor de simulación. El motor de simulación se comunica con el entorno, para poder realizar las simulaciones necesarias para las acciones de los agentes de la *Aplicación*. Parte de estas simulaciones necesitan de

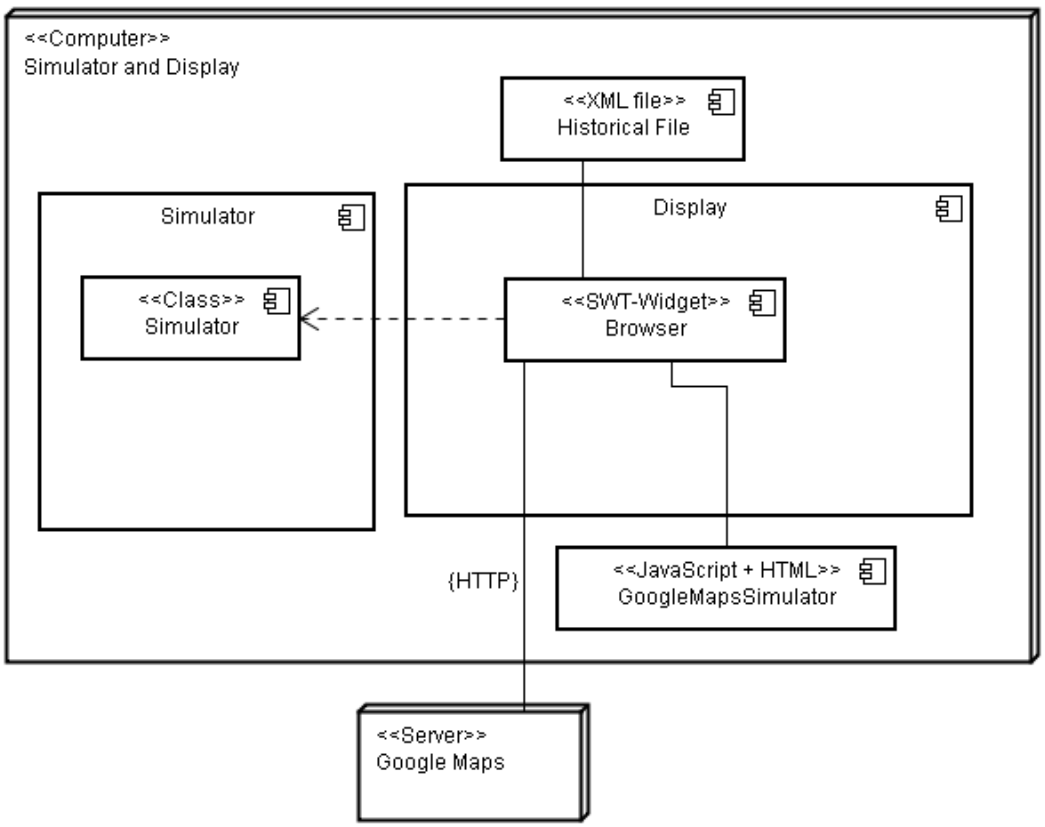


Figura 3.58: Diagrama de despliegue del subsistema *Visualizador*

un navegador Web que cargue el fichero llamado *GoogleMapsSimulator*, el cual, mediante una llamada previa al servidor de *GoogleMaps*, realizará los calculos relativos al movimiento. Además para la configuración de la simulación se utiliza el modulo *Setup*, por lo que el motor de simulación depende de este componente. El resultado de cada ejecución será almacenado por el módulo *Historical* en un fichero en formato XML, el cual se guardará en el mismo ordenador en el que se ha realizado la simulación.

3.6.1.3. Visualizador

El subsistema *Visualizador* se va a componer, entre otras cosas, de un componente visual de SWT llamado Browser (ver diagrama 3.58). Este navegador cargará la página Web alojada en el fichero *GoogleMapsSimulator* para mostrar la información de la simulación generada por el módulo *Simulador*. El *Visualizador* se encontrará en la misma máquina que está el *Simulador* para obtener los datos generados en el caso de que el usuario quiera verlos en tiempo real. Si más tarde el usuario quiere ver el contenido de un fichero histórico, éste fichero debe encontrarse en el mismo nodo que el *Visualizador*.

El navegador se comunicará con los servidores de Google Maps mediante el protocolo HTTP para cargar el mapa y manipular su contenido, según estén implementados los métodos JavaScript del fichero *GoogleMapsSimulator*.

3.6.2. Empaquetado

La implementación del sistema desarrollado se puede encontrar en el CD-ROM que acompaña a esta memoria dentro de la carpeta **mHealth**, cuyos componentes se distribuyen en ella de la siguiente manera:

- mHealth/bin: en este directorio se encuentran los ficheros *.class* creados mediante la compilación.
- mHealth/lib: contiene las librerias de las que depende el proyecto para ser compilado.
- mHealth/experiments: en este directorio se encuentran datos de entrada para la simulación o visualización.
    - mHealth/experiments/configurations: se incluyen ficheros de configuración para simulaciones.
    - mHealth/experiments/historical: se incluyen ficheros históricos de simulaciones previas para poder visualizarlas.
- mHealth/src/es/urjc/ia/at/mhealth: contiene los ficheros fuente del proyecto.
    - mHealth/application: código fuente relacionado con el subsistema *Aplicación*.
        - application/agents: contiene la clase abstracta *Participant* y los agentes que la extienden.
            - agents/internalStates: contiene los diferentes tipos de estados internos de cada agente.

        - ◇ agents/selectionMethods: contiene diferentes implementaciones de selección de recursos (centros de emergencias, ambulancias y hospitales).
    - application/ontology: contiene la ontología que utilizan los agentes y el entorno.

- mHealth/display: codigo fuente relacionado con el subsistema *Visualizador*.
    - display/gui: contiene aquellas clases Java que implementan las interfaces gráficas.
        - ◇ gui/images: contiene las imagenes necesarias para la *gui* del *Visualizador*.
    - display/information: contiene las clases necesarias para generar la información a visualizar.
    - display/player: contiene la clase que visualiza la simulación cuando la simulación es offline.
- mHealth/maps: código fuente común para el *Visualizador* y el *Simulador* sobre cálculos de rutas en mapas.
    - maps/googleMaps: ficheros HTML con el contenido necesario para la creación de mapas y métodos de manipulación.
        - ◇ googleMaps/images: contiene los iconos usados en los mapas de *GoogleMaps*.
- mHealth/simulator: codigo fuente relacionado con el subsistema *Simulador*.
    - simulator/agentsSimulator: aquí se encuentran las clases que simulan las acciones de los diferentes agentes de la *Aplicación*.

- simulator/environment: paquete que engloba las clases encargadas de la ejecución del agente Entorno.
  - environment/distributions: contiene las distribuciones de ocupación de camas, de como enferman los pacientes, de como se generan los pacientes y de los focos de población.
  - environment/maps: contiene la clase Java que implementa el mapa del Entorno mediante el uso de *GoogleMaps*.
- simulator/historical: contiene la clase que gestiona cada *step* de la simulación para posteriormente crear un histórico mediante la clase `Historical`.
- simulator/setup: Paquete que engloba la funcionalidad necesaria para crear la configuración de una simulación.
  - setup/configuration: contiene las clases que almacenan los parámetros de la configuración seleccionados por el usuario.
  - setup/gui: conjunto de clases visuales encargadas de solicitar al usuario los parámetros de la simulación necesarios.

3.6.3. Detalles de implementación

En esta sección, se detallan las decisiones tomadas sobre la implementación y los conceptos más importantes desarrollados en esta fase del proyecto:

3.6.3.1. Implementación de agentes con Jade

Como se ha dicho anteriormente, el subsistema *Aplicación* será implementado mediante el framework JADE. Esto nos permitirá crear agentes JADE a los cuales les añadiremos un comportamiento especifico para la recepción de percepciones y el envío de acciones.

Los motivos para utilizar esta plataforma de desarrollo de agentes son numerosos, algunos de estos motivos son los siguientes:

- JADE cumple con la especificación de FIPA, luego puede comunicarse con agentes realizados en otros entornos que sigan FIPA.
- JADE proporciona multiples mecanismos de transporte de mensajes, siendo transparentes para el programador, por lo que únicamente deben implementarse los mensajes ACL a enviar.
- Los agentes residen en un entorno de ejecución llamado plataforma la cual puede tener uno o mas contenedores. Los contenedores son instancias del entorno de ejecución de JADE en los que se encontrarán los agentes, además siempre existe un único contenedor principal algunos de los agentes creados por JADE tales como el AMS y el *Directory Facilitator* DF.
- Permite utilizar protégé para el desarrollo de ontologías. Además, es posible comunicar JADE con aplicaciones realizadas en java.

Cuando se desea crear un agente JADE se debe implementar una clase Java que herede de la clase `jade.core.Agent`, esta clase nos proporciona los elementos básicos para manejar el ciclo de vida de los agentes, ejecución de tareas y comunicación entre agentes utilizando mensajes ACL. Los agentes implementados a partir de la clase `jade.core.Agent` deben redefinir el método *`setup()`* ya que se utiliza para inicializar el agente. Básicamente lo que tendremos que hacer es registrar los lenguajes de contenido y ontologías que conoce el agente y añadirle sus comportamientos.

El registro de los lenguajes de contenido y ontologías se realiza a través del `ContentManager`. Se trata de una clase que poseen todos los agentes y que contiene los metodos necesarios para la transformación de objetos java de tal manera

que puedan ser utilizados como contenido de los mensajes ACL. Para poder realizar esta tarea, en el `ContentManager` los agentes deben registrar el *codec* del lenguaje de contenido y la ontología que utilizan. Una vez realizado el registro, la ontología se encargará de validar la información y el *codec* realizará la traducción a *string* de acuerdo a las reglas sintácticas del lenguaje de contenido.

Los agentes son capaces de realizar tareas mediante el modelado de comportamientos. JADE nos proporciona la librería `jade.core.behaviours` en la que podemos encontrar diferentes tipos de comportamientos o *behaviours*. Algunos de estos comportamientos son:

- OneShotBehaviour: se ejecuta una sola vez.
- *CyclicBehaviour*: debe ser ejecutado siempre.
- *TickerBehaviour*: periódicamente ejecuta una cierta tarea.

Como podemos observar el comportamiento *CyclicBehaviour* nos proporciona la funcionalidad que deseamos para los agentes de la aplicación ya que estos deben estar continuamente en contacto con el entorno para que, una vez recibidas las percepciones, puedan determinar qué acción realizar y enviársela al entorno. Por esta razón los agentes se han modelado utilizando este comportamiento y las tareas que se realizan en él son las siguientes:

- Recibir percepción: es un método que se encarga de recibir los mensajes ACL que envía el entorno y mediante el `ContentManager` se encargará de extraer el contenido del mensaje para poder acceder a la percepción.
- Procesar percepción: este método será implementado teniendo en cuenta el tipo de agente que estamos tratando, ya que se encarga de analizar la percepción para actualizar su estado interno en consecuencia.

- Seleccionar próxima acción: también este método es dependiente del tipo de agente, ya que deberá de analizar su estado interno para determinar que acción es la más adecuada en ese instante.

- Enviar acción: se encarga de componer un mensaje ACL cuyo contenido será la acción que el agente desea realizar y cuyo destinatario es el entorno.

Cabe destacar que todos los agentes de la aplicación realizarán las cuatro tareas que hemos definido, aunque, como hemos visto en diseño 3.14, nuestra aplicación se compone de miembros y organizaciones, pudiendo estas últimas enviar más de una acción por cada percepción recibida.

El código del método *setup()* de los agentes será el siguiente:

```
protected void setup() {
    getContentManager().registerLanguage(codecLenguajeContenido);
    getContentManager().registerOntology(ontologia);
    CyclicBehaviour behaviour = new CyclicBehaviour() {
        Percepcion percepcion = recibirPercepcion();
        procesarPercepcion(percepcion);
        AgentAction accion = seleccionarAccion();
        enviarAccion(accion);
    }
    addBehaviour(behaviour);
}
```

Una vez definido el comportamiento de los agentes vamos a describir como implementar una ontología en JADE. Para ello debemos tener en cuenta que una ontología en JADE es una instancia de la clase `jade.content.onto.Ontology` en la que se añaden los esquemas que definen la estructura de los tipos de predicados (`Predicate`), acciones (`AgentAction`) y conceptos (`Concept`) relevantes del dominio que se desee representar. Los predicados se utilizarán para relacionar

conceptos. Por ejemplo en nuestra aplicación existirá el predicado "Percibe" al que se le asociara una "Percepcion" y los conceptos son los definidos en 3.16. Esta ontología será el vocabulario común mediante el cual los agentes intercambiarán su conocimiento.

3.6.3.1.1. Decisiones de diseño para implementar el entorno como agente Jade

La arquitectura del proyecto permite tener módulos claramente diferenciados para evitar el acoplamiento. Por ello, aunque el subsistema aplicación este creado mediante el framework JADE, el simulador podrá ser implementado mediante otras tecnologías siempre y cuando el entorno sea capaz de comunicarse con los agentes de la aplicación. Para que esta comunicación pueda llevarse acabo hay que tener en cuenta como son los mensajes que recibe y envía la aplicación. Se ha tomado la decisión de implementar el entorno del simulador como un agente JADE ya que facilita la comunicación con los agentes compartiendo la misma ontología.

3.6.3.2. Representación informática del dominio de las emergencias médicas

Conseguir dar alto realismo a cada escenario de emergencias médicas que se simule en el demostrador *mHealth*, requiere entender muy bien el dominio que se está usando. Cada actor que participa en este escenario debe reflejar de forma correcta la entidad del mundo real que representa y sus comportamientos deben ser lo más reales posibles para que la simulación sea válida. Con ayuda del SUMMA de Madrid, se ha capturado suficiente información para poder entender mejor las características de cada entidad de este dominio, así como sus comportamientos y las interacciones entre ellas. Pero adecuar esa información a una aplicación informática, no ha sido una tarea sencilla. Las decisiones llevadas a cabo y los conceptos más importantes desarrollados se exponen a continuación.

3.6.3.2.1. Modelado de los actores

Como se explicó en la descripción del problema (sección 2.1), en el escenario de emergencias médicas detallado participan cuatro tipos de actores: pacientes, hospitales, ambulancias y centros de emergencias. Cada uno de ellos se caracteriza por tener un conjunto de parámetros llamado "estado interno", que nos muestra cómo se encuentra ese actor en un momento determinado. Además, cada actor puede realizar un número limitado de acciones en el entorno (enumeradas en la sección de Análisis, diagrama de casos de uso 3.3) y, en función de la acción que solicite hacer o de las acciones que realicen otros actores (algunas pueden influir en otros actores diferentes al que realiza la acción), su estado interno cambiará.

Los atributos que forman parte del estado interno de cada actor deben modelar en su conjunto a la entidad del mundo real que se quiere representar. Por ello, se han descrito sus estados internos de la siguiente forma:

- Centros de emergencia
    - Identificador: único para cada agente del entorno, de forma que se le pueda identificar inequívocamente en la plataforma de agentes.
    - Alias: nombre que se le da al centro de emergencias el usuario del sistema.
    - Entorno: identificador del agente entorno, para que el hospital sepa a quién debe mandarle las acciones. En la vida real, es como indicarle a la entidad en qué mundo está actuando.
    - timeStep: tiempo que tiene el centro para decidir qué acción hacer y enviársela al entorno.
    - Última acción: el centro guarda en su memoria la última acción que solicitó simular, para entender mejor lo que percibe después del entorno

en el que está.

- Número de operadores telefónicos: en a realidad, un centro coordinador de emergencias dispone de un número de operadores telefónicos que tramitan las llamadas que recibe el centro. Esto implica, que en un mismo momento el centro puede atender a varias personas a la vez, llamar a diferentes ambulancias en paralelo, etc. Indicando un número de operadores telefónicos a nuestros centros de emergencia, permitimos que el centro envíe una lista de acciones al entorno de tamaño máximo igual al número de telefonistas que le forman. Con este sistema, conseguimos que nuestros centros se comporten como organizaciones en vez de como agentes individuales.
- Localización: indica la posición geográfica de la central de emergencias.
- Información de las ambulancias: el centro debe tener un listado con la información que necesite conocer acerca de las ambulancias, incluyendo sus localizaciones, estados de misiones, etc. Esa información es muy importante en un centro de emergencias porque es fundamental para saber coordinar con éxito las misiones de urgencias existentes.
- Listado de llamadas entrantes de pacientes: el centro debe conocer el listado de todas las llamadas que le llegan de pacientes solicitando asistencia médica.
- Listado de llamadas de pacientes en espera: en esta lista almacena la información de aquellos pacientes que han llamado y están esperando una ambulancia.
- Listado de llamadas a ambulancias: por cada paciente que llama, el centro debe saber a qué ambulancias están llamando el resto de operadores telefónicos para asignar misiones a ambulancias libres.

- Listado de misiones activas: el centro debe conocer en todo momento el estado de cada misión que ha asignado a cada ambulancia, para saber cuáles están ocupadas, qué pacientes están siendo atendidos, etc.
- Listado de misiones fallidas: para realizar estadísticas, el centro almacena la información de cada misión realizada, de forma que puede conocer los motivos de porqué una misión no se realizó con éxito y así, poder mejorar en su coordinación.

- Pacientes
  - Identificador: único para cada agente del entorno, de forma que se le pueda identificar inequívocamente en la plataforma de agentes.
  - Alias: como los pacientes los genera el simulado, los alias se corresponderán con sus identificadores.
  - Entorno: identificador del agente entorno, para que el hospital sepa a quién debe mandarle las acciones. En la vida real, es como indicarle a la entidad en qué mundo está actuando.
  - timeStep: tiempo que tiene el paciente para decidir qué acción hacer y enviársela al entorno.
  - Última acción: el paciente guarda en su memoria la última acción que solicitó simular, para entender mejor lo que percibe después del entorno en el que está.
  - Localización: indica la posición geográfica del paciente en cada momento.
  - Estado físico: sus posibles valores son *sano*, *enfermo* y *muerto*. De esta forma, diferenciamos a un paciente cuando necesita asistencia médica

de cuando se ha curado gracias a una adecuada asistencia médica o ha fallecido por una mala gestión del sistema de urgencias.

- Estado del paciente: un paciente puede estar *llamando* o *en espera*. El primer estado, indica que el paciente está solicitando asistencia al centro coordinador y el segundo, indica que el paciente está a la espera de la asistencia médica.
- Estado de emergencia: cuando un paciente no está ni llamando a un centro ni en espera, es porque le están asistiendo. En ese caso, sus posibles estados son *en ambulancia*, *en hospital* y *recibiendo asistencia in-situ*. En el primero, el paciente está dentro de una ambulancia, de forma que ésta le intenta mantener estable mientras le traslada a un hospital. En el segundo estado, el paciente ha sido trasladado e ingresado al hospital. En el tercero, el paciente está siendo atendido por una ambulancia *in-situ*.
- Nivel de enfermedad: Dado que modelar los síntomas de una enfermedad es complejo, se ha implantado un sistema de niveles de enfermedad para los pacientes parecido al que utiliza el SUMMA: cada conjunto de síntomas se agrupa en un número, del 1 al 5, de forma que un paciente leve tendrá un nivel de enfermedad bajo y uno más grave, lo tendrá más alto. Con estos niveles, el centro coordinador sabrá que recurso es el adecuado para la asistencia a ese paciente y cual no lo es.
- Listado de los centros de emergencia: el paciente tiene un listado con los identificadores de los centros de emergencia que existen en su entorno. Simula el hecho de tener un listado de números de teléfono a los que llamar, como por ejemplo, el 112 del SUMMA de Madrid.

- Ambulancias

- Identificador: único para cada agente del entorno, de forma que se le pueda identificar inequívocamente en la plataforma de agentes.
- Alias: nombre que se le da a la ambulancia el usuario del sistema.
- Entorno: identificador del agente entorno, para que el hospital sepa a quién debe mandarle las acciones. En la vida real, es como indicarle a la entidad en qué mundo está actuando.
- timeStep: tiempo que tiene la ambulancia para decidir qué acción hacer y enviársela al entorno.
- Última acción: la ambulancia guarda en su memoria la última acción que solicitó simular, para entender mejor lo que percibe después del entorno en el que está.
- Localización: indica las coordenadas geográficas de la ambulancia en cada momento de la simulación.
- Información del paciente asignado: cuando un centro asigna un paciente a una ambulancia, ésta recibe la información que el paciente dio a priori al centro. Una vez la ambulancia esté con el paciente, esa información se actualizará simulando la percepción del estado del paciente por parte de la ambulancia.
- Capacidad de su equipo médico: Para modelar que existen varios tipos de ambulancias, se le definen a cada una un conjunto de parámetros que la dotan de mejor o peor equipo, compuesto de profesionales médicos y herramientas sanitarias, para cada tipo de nivel de enfermedad que un paciente puede tener. De esta forma, se puede tener una ambulancia con un 100 % de equipación para un nivel 1 de enfermedad pero un 10 % de equipación para un nivel 5. Esto querrá decir que si asignan a la ambulancia un paciente con enfermedad 1, le podrá curar con

alta probabilidad, ya que dispone de los recursos necesarios para la asistencia de ese nivel. Pero si le asignan un paciente con enfermedad 5, apenas podrá hacer nada por él en la asistencia *in situ*, ya que no posee un equipo adecuado para curarle.

- Ruta de recorrido: igual que en la vida real, una ambulancia puede tener una zona de actuación por la que va moviéndose en caso de no tener activa ninguna asistencia, o puede estar aparcada en un aparcamiento de ambulancias esperando una misión. Para ambos casos, se le indica a la ambulancia un conjunto de puntos geográficos (o un único punto en el caso de un aparcamiento) donde la ambulancia puede estar cuando está libre.
- Listado de los hospitales del entorno: para saber a qué hospital puede llevar la ambulancia a un paciente, ésta debe conocer qué hospitales forman parte de su entorno de actuación y así, poder llamarles para solicitar el ingreso de un paciente.
- Ocupación: este estado tiene dos posibles valores, *ocupado* y *disponible*. Cuando una ambulancia acepta una misión, pasa de disponible a ocupada. Y cuando esa misión finaliza, vuelve a ponerse disponible. De esta forma, el centro coordinador conoce a qué ambulancias puede llamar para asignarlas misiones y a cuales no.
- Estado de la ambulancia: una ambulancia puede estar *moviéndose*, *atendiendo a un paciente in situ*, *parada* o *esperando misión*. Una ambulancia no puede estar en más de un estado a la vez, por lo que el conjunto de estos estados define el comportamiento de una ambulancia. Dentro del primer estado, una ambulancia podrá estar moviéndose hacia un paciente, hacia un hospital o libremente. Esa información se recoge en el estado de la misión, ya que si ese estado es vacío, quiere

decir que la ambulancia está disponible y se está moviendo por su ruta de recorrido.

- Estado de la misión: puede tener los valores *llamando*, *yendo hacia un paciente* o *yendo hacia un hospital*. Cada uno de estos valores se corresponde con las posibles fases que tiene una misión de asistencia de urgencias. Cuando una ambulancia comienza una misión, tomará el segundo estado. Tras deliberar que un paciente necesita el ingreso en un hospital, la ambulancia llama al hospital seleccionado y se coloca en el primer estado. Cuando un hospital acepta el ingreso de un paciente, la ambulancia le transporta a ese hospital y adopta el tercer estado.
- Destino: cuando una ambulancia se está moviendo, por el motivo que sea, indica hacia dónde se mueve. Ese lugar, se almacena en este atributo para que la ambulancia tenga memoria de hacia dón de se está desplazando.
- Información de la misión asignada: cuando una ambulancia acepta una urgencia, recibe una serie de datos, como el estado del paciente, su localización, etc. Esta información se va actualizando según se suceda la misión.

- Hospitales

  - Identificador: único para cada agente del entorno, de forma que se le pueda identificar inequívocamente en la plataforma de agentes.
  - Alias: nombre que se le da al hospital el usuario del sistema.
  - Entorno: identificador del agente entorno, para que el hospital sepa a quién debe mandarle las acciones. En la vida real, es como indicarle a la entidad en qué mundo está actuando.

- timeStep: tiempo que tiene el hospital para decidir qué acción hacer y enviársela al entorno.
- Última acción: el hospital guarda en su memoria la última acción que solicitó simular, para entender mejor lo que percibe después del entorno en el que está.
- Localización: ubicación física del hospital en el entorno.
- Personal de urgencias: igual que pasaba con los centros coordinadores, un hospital se compone de un número de personas que trabajan en el área de urgencias. Según sea ese número de personas, el hospital tendrá mayor capacidad de atención a los pacientes o menor. Indicando el número de profesionales que trabajan en urgencias, podemos permitir que el hospital funcione como una organización donde tiene permitido realizar tantas acciones en un mismo momento como personal tenga trabajando.
- Número de camas: indica el número de camas que tiene el hospital, libres u ocupadas.
- Número de camas ocupadas: para conocer la tasa de ocupación del hospital, se indica el número de camas ocupadas que tiene en cada momento. Este dato fluctuará según la llegada de pacientes desde ambulancias, así como pacientes que llegan por libre o desde traslados de otros hospitales. Estos dos últimos casos, forman parte de los dos escenarios de emergencias médicas que no se han tratado en este simulador, explicados en la introducción de la memoria.
- Nivel de urgencias: a los hospitales se les añade un parámetro que define su nivel de urgencias máximo, valor entre 1 y 5 (en correspondencia con los niveles de enfermedad de pacientes) para indicar hasta

qué nivel es capaz el área de urgencias del hospital de atender con éxito a los pacientes que llegan a él.

- Listado de llegadas de pacientes: el hospital será consciente en todo momento de los pacientes que llegan a su puerta con esta lista. Aquí sólo se reflejan los pacientes que llegan a través de ambulancias, las cuales deberán haber hecho una reserva previa del ingreso del paciente.
- Información de los pacientes ingresados: para conocer el estado de salud de cada paciente ingresado, se almacena su información en una lista y se irá actualizando según la evolución del paciente dentro del hospital.Una vez que se le de el alta, o que el paciente fallezca, se elimina de esta lista y el hospital deja de percibirlo.
- Reservas desde ambulancias: como anteriormente se ha comentado, las ambulancias llaman a los hospitales preguntando si aceptan el ingreso de un paciente. Si estas contestan en afirmativo, deben hacer una "reserva" ficticia de forma que el hospital no llegue a una ocupación del 100 % sin haber admitido al paciente que va a llegar desde la ambulancia aún, quedando al menos libres tantas camas como reservas haya.

3.6.3.2.2. Distribuciones estadísticas

Conocer cuántos pacientes llaman a un centro de emergencias en un cierto intervalo de tiempo, o qué puntos geográficos concentran mayor número de ciudadanos, son datos que a veces no se pueden conocer con certeza pero sí se pueden estimar mediante distribuciones estadísticas. Se han definido una serie de distribuciones en el simulador para dotar de aleatoriedad a ciertos comportamientos, entre las cuales el usuario puede elegir las que mejor representen el escenario de emergencias médicas que quiere configurar en el módulo *Setup*.

- Generación de pacientes: los agentes que representan a los pacientes son los únicos que se van generando durante la simulación, en vez de crearse al comienzo como sucede con el resto de actores. Esto se debe a que los pacientes sólo aparecen en el entorno cuando se ponen enfermos y desaparecen cuando se curan o mueren, para no saturar la plataforma de agentes. La creación de pacientes, por tanto, se corresponde con la distribución que sigue el número de llamadas recibidas en un centro de emergencias, ya que según se generen, llamarán al centro solicitando asistencia. Este concepto se ha modelado con las siguientes distribuciones:

  - Uniforme: en la configuración se indica el número total de personas que se pondrán enfermas durante la simulación y el número de pasos totales que formarán la simulación. Con esos datos, se puede calcular el número de pacientes que debe aparecer en cada *timestep* de manera que se repartan los pacientes entre los intervalos de tiempo uniformemente.
  - Todos al principio: cuando se quiere ver qué ocurre si en un escenario hay un número determinado de pacientes que llaman al centro de emergencias en el mismo momento, se crea una distribución que, simplemente, genera a todos los pacientes enfermos en el primer *timestep* de la simulación. Durante el resto de ella, no se generarán más pacientes, solamente se verá cómo va evolucionando la asistencia médica de esos enfermos iniciales hasta el final de la simulación.
  - Un paciente en cada *timestep*: parecida a la distribución anterior, se ha creado ésta en la que todos los pacientes se irán generando en los primeros pasos de la simulación de uno en uno, para hacer que aparezcan en el entorno en los primeros *timesteps* pero de forma escalonada. Así, se puede ir viendo cómo rápidamente el centro coordinador se satura al acumular llamadas de enfermos y cómo debe repartir sus

recursos en ese caso.

- Niveles de enfermedad: cuando se generan los pacientes, se deben indicar los valores de los atributos que forman sus estados internos. Uno de estos valores es el nivel de enfermedad. Tal y como se explicó anteriormente, este atributo se ha modelado con un número comprendido entre 1 y 5 (este último nivel podrá ser más pequeño, acortando el intervalo de enfermedades, si el usuario lo desea). Estos números se corresponden con enfermedades más leves o más graves, respectivamente. El nivel de enfermedad que tendrá cada paciente se puede modelar con las siguientes distribuciones:

  - Uniforme: un paciente puede tener de manera equiprobable cualquier nivel de enfermedad definido en el sistema. De esta forma, las enfermedades se repartirán de manera uniforme por todos los pacientes del entorno.
  - Exponencial: un paciente tendrá más probabilidad de tener una enfermedad leve que una muy grave, imitando a la vida real. Los niveles intermedios de enfermedad seguirán una distribución exponencial de manera que a mayor gravedad de la enfermedad, menor será la probabilidad de que el paciente presente esos síntomas.

- Localización de los pacientes: otro de los parámetros que se deben definir en el estado interno de un paciente es su localización inicial. Esta será la localización que le indicará al centro coordinador para decirle dónde debe recogerle la ambulancia. Dicho atributo viene dado por las siguientes distribuciones estadísticas:

  - Uniforme: en base a los límites del mapa del entorno donde actuarán los agentes, se calcula un punto geográfico calculando su latitud y longitud de forma aleatoria, de manera que cada posible punto geográfico

que se encuentra dentro de esos límites, tenga la misma probabilidad de ser seleccionado como localización de un paciente que el resto de puntos. De esta forma, se reparten a los pacientes enfermos por todo el territorio del entorno uniformemente.

- Densidad de población: el usuario, en el módulo *Setup*, puede definir su propia distribución de población definiendo puntos en el mapa con alta densidad demográfica, en los cuales aparecerán los pacientes con mayor probabilidad que en el resto del entorno. Dentro de esos puntos demográficos, se repartirá a los pacientes uniformemente.

- Ambulancias aleatorias: cuando un usuario selecciona en el módulo *Setup* que desea generar ambulancias en el entorno de forma aleatoria, los parámetros que la definirán seguirán todos una distribución uniforme. De esta forma, la capacidad de la ambulancia para cada nivel de enfermedad tendrá un valor entre 0 y 100 con misma probabilidad de selección. Y su localización, se calculará según los límites del mapa del escenario médico impuestos previamente, de forma que se escogerá un punto geográfico al azar, obteniendo primero la latitud y luego la longitud uniformemente entre todos los posibles valores.

- Hospitales aleatorios: con los hospitales aleatorios sucede igual que con las ambulancias aleatorias. Los parámetros que definen a un hospital, como son su localización, capacidad de urgencias y número de camas, se seleccionan siguiendo una distribución uniforme para que los diferentes valores posibles en cada atributo sean equiprobables.

- Ocupación de los hospitales: como ya se mencionó anteriormente, se han modelado los hospitales de forma que no sólo reciban pacientes desde el transporte de ambulancias, sino que se quiere que se simulen llegadas de

pacientes a pie o trasladados desde otros hospitales. Como estos casos no forman parte del escenario elegido, no se han detallado pero sí se han añadido en forma de distribución estadística, dotando de aleatoriedad a la ocupación de las camas de urgencias de cada hospital. Este comportamiento se ha modelado con las siguientes distribuciones:

  - Uniforme: en cada *timestep* de la simulación, la ocupación del hospital se procesará obteniendo un valor equiprobable entre 0 y el número máximo de camas existentes en el hospital una vez restadas el número de camas ocupadas por pacientes llegados desde ambulancias. De esta forma, se respeta la ocupación generada por casos simulados en nuestro entorno, dejando la aleatoriedad al número de camas ocupadas desde otros casos de urgencia.
  - Normal: el punto álgido de ocupación llegará en torno a la mitad del tiempo de simulación. Al igual que anteriormente, se respetará la ocupación de aquellas camas con pacientes llegados desde ambulancias. La tasa de ocupación irá creciendo de forma exponencial hasta la mitad del tiempo total de la simulación, y luego irá disminuyendo.

- Tiempos de vida y muerte de pacientes: un paciente tiene un nivel de enfermedad asociado y, según sea ese nivel, se le asigna un tiempo de vida proporcional. Ese tiempo se irá reduciendo según pasen los *timesteps* de la simulación y si llega a 0, el paciente morirá. Para emular que un paciente se puede curar, se le asigna otro tiempo en función del nivel de enfermedad que tenga: un tiempo de curación. Este parámetro aumentará según reciba asistencia médica y si consigue llegar al máximo del tiempo de curación, se dará al paciente por sano. Pero el aumento del tiempo de curación no será uniforme, dependerá de factores como el tipo de asistencia que reciba

y la adecuación del equipo médico que le atienda, definiendo dos tipos de asistencias: positivas, en las que se aumenta el tiempo de curación y de vida del paciente y negativas, en las que el tiempo de vida disminuye y el de curación no varía.

Los tipos de asistencia se corresponden con los estados por los que transita un paciente a lo largo de su vida en la simulación y se clasifican en los siguientes puntos:

- El paciente espera una ambulancia: en este estado, el paciente no recibe aún asistencia médica. Por ello, su tiempo de curación permanece en 0 y el tiempo de muerte va disminuyendo según pasan los *timesteps*.
- El paciente recibe asistencia *in situ* de una ambulancia: en este momento, el paciente recibe el primer tipo de asistencia médica que puede llegar a curarle. Una asistencia tendrá una duración de *n-timesteps*, según decida la propia ambulancia que hace la asistencia. En cada uno de esos *timesteps*, se calcula la probabilidad de que sea una asistencia positiva o negativa según esté definido el atributo "capacidad" de la ambulancia (la adecuación del equipo que compone la ambulancia a cada nivel de enfermedad): a mayor capacidad de la ambulancia, mayor probabilidad de que la asistencia sea positiva. Se obtiene un número aleatorio entre 0 y 100. Si ese valor es mayor que la capacidad, la asistencia es negativa y se decrementa el tiempo de vida del paciente, simulando un empeoramiento de su estado. Si ese valor es menor que la capacidad, se pasa a determinar si la asistencia es negativa o positiva en base al nivel d enfermedad del paciente: a mayor gravedad del paciente, menor es la probabilidad de curarse. Se obtiene un número aleatorio entre 0 y el número máximo de enfermedad. Si

ese valor es menor que el nivel del paciente, la asistencia es negativa y se decrementa su tiempo de vida. Si ese valor es mayor, la asistencia es positiva y se le añade un punto al tiempo de curación además de aumentar el tiempo de vida del paciente, simulando una mejora en su estado de salud.

- El paciente es trasladado a un hospital: si un paciente no llega a curarse con la asistencia *in situ*, la ambulancia le trasladará a un hospital. Durante ese traslado, consideramos que el paciente recibe un tipo de asistencia en la que su gravedad no mejora, pero si puede estabilizarse. Para modelar esta idea, se sigue el siguiente procedimiento: el paciente estabilizará su estado con probabilidad igual a la capacidad de la ambulancia asociada a su nivel de enfermedad. Para ese cálculo, se obtiene un número aleatorio entre 0 y 100. Si ese número es mayor que la capacidad, la asistencia en el traslado es negativa y la vida del paciente disminuye. Si ese número es menor que la capacidad, sus tiempos de vida y muerte no varían. Con este método, se consigue simular que el paciente se estabiliza y, en el caso de que el paciente siempre recibiese asistencias positivas en el traslado hacia un hospital, nunca llegaría a curarse, sólo se quedaría estable hasta entrar a ese hospital.

- El paciente ingresa en un hospital y recibe asistencia médica: en este caso se procede de forma similar a la asistencia *in situ* de una ambulancia. El hospital dará asistencias positivas o negativas en función del nivel máximo de urgencias que tenga. Si el nivel de urgencias del hospital es menor que el nivel de enfermedad del paciente, siempre la asistencia será negativa, puesto que el hospital no es el adecuado para su curación. En caso contrario, se calcula la probabilidad de que el paciente se cure en base al nivel de enfermedad que tenga: a mayor

gravedad del paciente, menor es la probabilidad de curación. Se obtiene un número entre 0 y el nivel máximo de enfermedad. Si ese número es mayor que el nivel de paciente, la asistencia es positiva y se aumentan sus tiempos de vida y curación para simular mejora en el paciente. Si ese numero es menor, se decrementa su tiempo de vida, simulando una asistencia negativa.

3.6.3.3. Decisiones de diseño para implementar el Visualizador

Las partes más importantes del desarrollo del subsistema *Visualizador* han sido las siguientes:

3.6.3.3.1. Google Maps

Un requisito que debía cumplir la aplicación era la simulación de movimientos de los agentes de la forma más real posible. Esto ha sido posible gracias al uso de la API de Google Maps, la cual proporciona diversas utilidades para manipular mapas, calcular rutas y añadir contenido al mapa. En nuestro proyecto, se ha utilizado la versión 2.0 de la API que incluye, entre otras, las siguientes clases:

- GMap2: clase central de la API. Es la encargada de crear un mapa dentro del contenedor HTML que se le indique en el constructor del objeto.
- GLatLng: representa una coordenada geográfica en el mapa, definida por una latitud y una longitud en grados (latitud desde -90 grados hasta +90 grados y longitud entre -180 grados y +180 grados).
- GIcon: se utiliza para cargar una imagen específica como icono en el mapa. Este icono deberá posicionarse en el mapa a través de un marcador, llamado `GMarker`.

- GMarker: marca una posición en el mapa, especificando sus coordenadas con un objeto GLatLng y visualizándola a través del icono `GIcon` que se le indique al construirse.

- GPolygon: superposición del mapa que dibuja un polígono, indicando varios puntos de coordenadas que representarán los vértices del polígono, los cuales se unirán visualmente mediante líneas rectas.

- GDirections: clase usada para obtener cálculos de rutas, desde un origen hasta un destino, y mostrarlos en el mapa cargado o en paneles de información en formato de texto, o en ambos.

- GRoute: los objetos de esta clase son creados por `GDirections`. Almacenan la información sobre una ruta única entre un origen y un destino.

- GEvent: contiene funciones que se usan para el registro y control de eventos sobre el mapa, ya sean personalizados por el programador o eventos DOM.

Para visualizar mapas y trabajar con ellos, es necesario el uso de un navegador web que interprete el lenguaje JavaScript con el que esta desarrollada la API. A continuación, se explican algunos detalles de este lenguaje de script y del navegador usado para mostrar los mapas en nuestro sistema:

- Browser desde swt: Un mapa de Google Maps sólo puede ser visualizado a través de un navegador Web compatible. El sistema desarrollado es una aplicación de escritorio, por lo que ha sido necesario empotrar en la interfaz visual algún tipo de componente que nos permita explorar páginas Web.

  EL problema con el que nos hemos encontrado en un principio ha sido que, no sólo se querían visualizar mapas, sino que desde el código Java se debía poder manipular el contenido del mapa y viceversa: que desde el código

JavaScript cargado en el navegador pudiese llamar a clases Java de nuestra aplicación.

Este requisito nos obligó a tener que usar la API SWT (*Standard Widget Toolkit*), la cual proporciona un componente llamado "*Browser*" que permite crear un navegador en la interfaz de usuario y cargar en él páginas Web. Este componente tiene el método *execute(<JavaScript-function>)*, método que permite llamar a funciones JavaScript contenidas en el código de la página Web que está cargada en el navegador SWT.

Con el navegador que nos proporciona SWT ya tenemos completada la manipulación del mapa de Google Maps a través de clases Java, ya que solo habrá que pasar por argumentos al método *execute(<JavaScript--function>)* las llamadas a las funciones que hayamos implementado en los ficheros HTML que usan la API de Google Maps.

Para obtener la parte inversa, en la que necesitamos que la propia página Web ejecute métodos de clases Java, es necesario tener la versión de SWT 3.5. En esta versión del toolkit existe la clase *BrowserFunction*, la cual facilita esta funcionalidad. En el diagrama 3.59 se muestra el funcionamiento de la comunicación bidireccional entre clases Java y métodos JavaScript:

Las interfaces de usuario IUSetupSimulation,IUSetupEmergencyCenter, IUSetupAmbulances y IUSetupHospitals contienen un navegador Web, instancia de Browser, que cargará el fichero *GoogleMapsSetup.htm*. Tal y como se ha explicado anteriormente, a través del método del navegador "execute", estas clases podrán llamar a funciones JavaScript de ese fichero. Con esta comunicación se consigue que en el código HTML se muestren datos que son usados o generados por clases Java.

Las clases MapBounds, MapFinished, MapPopulation y RandomRoute son

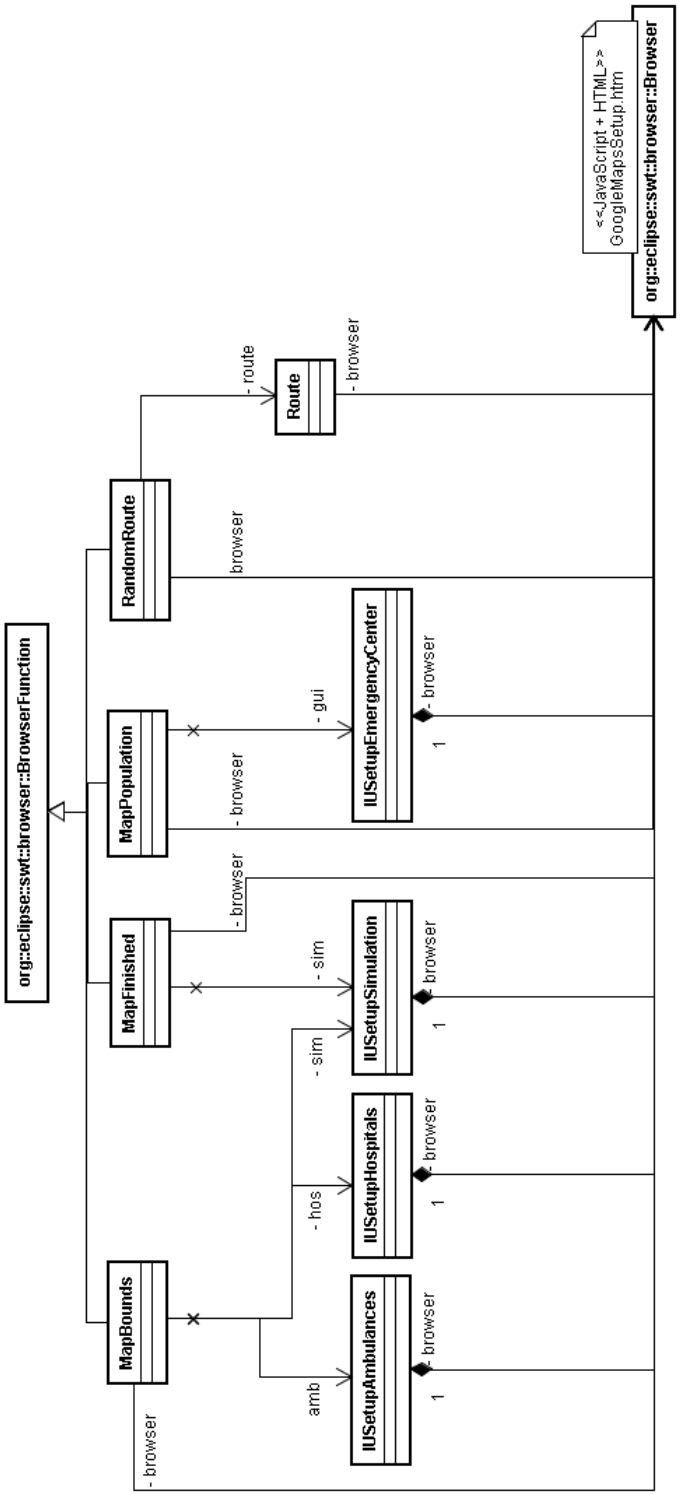


Figura 3.59: Diagrama general de clases de la interacción con JavaScript

aquellas que heredan los métodos de la clase BrowserFunction de SWT. Para que un método JavaScript pueda llamar a código Java, todas esas clases deben contener el método *`public Object function (Object[] arguments)`*, añadiendo el código que se desee ejecutar en su interior. Para que un método JavaScript sepa cómo se llama la clase Java que contiene *function*, se le debe pasar por parámetros al constructor de cada clase hija de BrowserFunction: el navegador que contiene el JavaScript desde el que se permitirá llamar a métodos Java y el nombre con el que se quiere asociar esa llamada a la clase en cuestión. De esta forma, se podrán usar en Java datos generados por el mapa de Google cargado en el navegador SWT.

- JavaScript: para la interacción del sistema con mapas de Google Maps se han creado dos páginas Web en ficheros HTML que contienen los métodos JavaScript y etiquetas HTML necesarias para el funcionamiento requerido. Estos ficheros, *GoogleMapSetup.htm* y *GoogleMapSimulator.htm*, son usados para la creación de la configuración mediante el *Setup* y para la simulación y visualización de movimientos, respectivamente. Ambas páginas Web contienen solamente un elemento HTML que albergará el mapa creado con la clase `GMap2`. Al cargarse la página, se creará ese mapa y se visualizará en aquellos navegadores que sean compatibles, es decir, aquellos que cumplan la condición *`GBrowserIsCompatible()`*, proporcionada por la API. Entre todas las funcionalidades implementadas en ambos ficheros, a continuación se explican las más importantes de cada uno:

  - Limitar el mapa del entorno dibujando un rectángulo: para especificar el área geográfico que contendrá el escenario de emergencias médicas de una simulación, en el *Setup* se debe poder elegir ese área sobre el dibujo de un mapa. Cuando el navegador cargue la página Web con

el mapa, se llamará a la función *editablePolygon()* del JavaScript que contiene. Esta función, registra un evento de la clase `GEvent`, que ejecuta la función *setPointMap()* cuando se hace click sobre el mapa. Este método, coloca dos marcadores *GMarker* en la posición del mapa pinchada, desactiva el evento anterior, y registra uno nuevo que se encarga de detectar si alguno de los marcadores anteriores esta siendo desplazado por el usuario sobre el mapa. En ese caso, se empieza a dibujar un rectángulo en el mapa, formado por el área que queda cubierta entre el marcador estático y el que e está desplazando. El polígono creado, de la clase *GPolygon*, se completará cuando se deje de desplazar el icono del marcador. Esto se recoge también con un *GEvent*, y en ese momento, se enviará al código Java una señal de que el mapa se ha completado, mediante la llamada *JavaFinishMap()*.

- Seleccionar puntos de alta densidad de población: cuando se selecciona una distribución de población nueva, el usuario puede definir en un mapa puntos de alta densidad de población, mediante el dibujo de unos círculos sobre dicho mapa. Para ello, cuando el usuario selecciona esa opción en el módulo *Setup*, se llama a la función *drawPopulation()* de la página Web cargada en el navegador. Este método es análogo al dibujo de los límites del mapa, pero en vez de un rectángulo, se dibujarán círculos, usando las funciones trigonométricas sobre circunferencias. Después de seleccionar un punto de población, si el usuario pulsa el botón de la interfaz que borra la distribución seleccionada, se llamará a la función *removePopulation()* para eliminar todas las marcas del mapa y poder empezar a dibujar de nuevo.
- Indicar posiciones de objetos en el mapa: tanto para las ambulancias como para los hospitales, se deben elegir ubicaciones de ellos

sobre el mapa del navegador. En el caso de las ambulancias, éstas pueden tener más de una localización, así que se identifica cada punto seleccionado del mapa con un icono diferente. Mediante la función JavaScript *`selectLocations()`*, se registra un evento para detectar cuando el usuario hace click sobre el mapa y en ese caso, se crea un `GMarker` en ese punto, se obtiene el identificador del marcador y sus coordenadas, y se devuelven al código Java a través de la llamada *`JavaGetBounds(letter, latlng.toUrlValue())`*. "latlng" es la variable que guarda las coordenadas, y para que en Java se entienda ese objeto, se pasa a un formato legible con la función de la clase `GLatLng` "toUrlValue". Respecto a los hospitales, se procede de la misma forma salvo que sólo puede tener una localización, por lo que usando el método JavaScript *`selectLocation()`* se registra un evento de selección del mapa que se desactivará en cuanto se produzca una sola vez. Mediante la clase de la API `GClientGeocoder`, se puede obtener la dirección en formato legible de las coordenadas donde se pinchó. De esa forma, cuando se elija una posición del hospital, se enviarán los datos al código Java con *`JavaGetBounds(response.name, place.address)`*, donde: "response.name" son las coordenadas geográficas y "place.address" es la dirección en formato de cadena legible para las personas. Finalmente, al indicar que el marcador del hospital es desplazable con el atributo "*draggable:true*", se registra otro evento para poder mover el icono del marcador hasta la posición deseada. Como en cada desplazamiento se vuelve a llamar al `GClientGeocoder`, se obtendrá siempre en el código Java la información asociada a la nueva posición del hospital.

- Calcular movimientos y representarlos: la API de Google Maps propor-

ciona la clase `GDirections` para calcular movimientos, pero no existe ningún método que permita representarlos directamente, sino que se debe calcular la velocidad del movimiento solicitado y los tramos en los que se divide la ruta para saber la posición exacta de un marcador en cada momento. Lo primero que debe hacerse es instanciar un objeto `GDirections` (en nuestro ejemplo, "directions[IDagent]") y llamar a su método *`load()`*, al cual se le tiene que indicar el punto de origen y el destino, en forma de texto que contenga las coordenadas greográficas de ambos puntos. Además se puede indicar, opcionalmente, parámetros sobre el idioma preferido en caso de que se quiera mostrar la información de la ruta en un panel textual. Para nuestro objetivo, usaremos la siguiente llamada: *`directions[IDagent].load(``from: '' + fromAddress + `` to: '' + toAddress,``locale'': ``es'', getPolyline:true, getSteps:false);`*. Con *getPolyline* indicamos que podremos obtener la polilínea del tramo a recorrer si llamamos al método *`directions[IDagent].getPolyline();`*. Esta polilínea consiste en la línea que dibuja la ruta por la que se pasará desde el origen al destino y contiene la siguiente información: el número de tramos rectos que forman la polilínea, el número de vértices que unen esos tramos rectos, la longitud total y la de cada tramo en metros, el tiempo medio que se tarda en recorrer toda la ruta y cada tramo de ella en segundos. Todos estos parámetros, se utilizarán para calcular los movimientos. El otro parámetro de *`load()`*, *getSteps*, se usa para obtener la información textual de la ruta para colocarla en panel de información. Al no usar dicho panel, ponemos esa opción a *false*. Una vez hecho esto, si se ha encontrado el origen y el destino y existe ruta entre ambos, se activará un evento "load" sobre el objeto `GDirections`. Con un `GEvent` capturamos dicho evento y comenzamos a calcular los movimientos con el método implementado *`Move()`*. A esta función se le indica el identificador del agente que va a realizar el movimiento (para poder mover a varios agentes a la vez sin conflictos), la distancia recorrida, la escala de tiempo con la que debe corresponderse el movimiento (definida en la configuración de la simulación) y la dirección destino. En nuestro

sistema, se quiere mover los marcadores con una precisión de 5 metros, la cual se ha guardado en la variable global *step*. Con esos datos, el método *Move()* implementa el siguiente algoritmo:

```
function Move(IDAgent, distancia-recorrida, escalaTiempo, destino) {
    distancia-total <- distanciaTotalRuta(directions[IDAgent])
    IF (distancia-recorrida >= distancia-total){
        nueva-posicion <- destino
        IDAgent.ponerEnPosicion(nueva-posicion)
        JavaSetLocation(IDAgent, nueva-posicion)
        eliminar(directions[IDAgent])
    }
    ELSE {
        nueva-posicion <- nextStep(IDAgent, distancia-recorrida)
        IDAgent.ponerEnPosicion(nueva-posicion)
        JavaSetLocation(IDAgent, nueva-posicion)
        IF (num-paso < numeroPasosEnRuta(directions[IDAgent]))
        AND (tramoNuevo(directions[IDAgent], num-paso)){
            tiempo <- calcularTiempoStep(
                tiempoDelTramo(directions[IDAgent], num-paso),
                distanciaDelTramo(directions[IDAgent], num-paso),
                step)
            tick <- calcularTiempoParaMoverse(step,escalaTiempo)
            num-paso <- num-paso+1
        }
        sleep(tick)
        Move(IDAgent, distancia-recorrida, escalaTiempo, destino)
    }
}
```

La función auxiliar *nextStep()* tiene la clave para colocar el marcador en la posición que le corresponde. Se basa en tomar el punto inicial

de la ruta como el metro 0, el punto final como el metro igual a la longitud total, y busca el tramo de la ruta que contiene al metro en el que se tiene que posicionar el marcador (distancia-recorrida). Una vez tiene ese tramo, obtiene las coordenadas de los vértices que contienen a esa línea recta y calcula la inclinación de esa recta para saber las coordenadas exactas que le corresponden a la distancia recorrida. Otra función importante es la que calcula la velocidad del tramo. Google Maps nos proporciona cierta información para poder calcular esa velocidad, que no es más que la distancia del tramo entre el tiempo que se tarda en recorrerlo. La técnica usada consiste en bloquear la llamada al método `Move()` el tiempo que debería tardar un coche en moverse los 5 metros del *step*. Ese tiempo se calcula en base a la velocidad del tramo donde se encuentra y de la escala del tiempo indicada por el usuario. Esta técnica funciona bien cuando la escala de tiempo es pequeña. Cuando la escala es muy grande, se pierde precisión, ya que llega un momento en el que se debería tardar en recorrer 5 metros menos de 1 milisegundo (tiempo mínimo que puede bloquearse la función), por lo que no funcionaría el bloqueo y se movería siempre a una velocidad constante. El simulador conocerá las posiciones de los agentes en cada momento mediante la llamada al código Java `JavaSetLocation`, la cual comunica la posición nueva de cada agente en cada nuevo cálculo de su movimiento.

- Mostrar la información de cada agente del entorno: en cada paso de la simulación, los estados internos de los agentes que forman el entorno médico se van modificando, según las acciones que soliciten hacer. Algunos datos de esos estados internos son mostrados de forma gráfica, o bien en el panel de información de la interfaz gráfica SWT, o en el mapa

cargado en el navegador de dicha interfaz. Para que se puedan mostrar en el mapa, el código Java debe llamar a la función JavaScript de la página Web ***drawSimulationStep(json)*** , pasándole por parámetros un String con toda la información necesaria de los agentes en formato JSON (a través de la librería *XStream*). Cuando se hace esa llamada, el método JavaScript transforma ese String JSON en un objeto manipulable por JavaScript a través de la función ***eval()*** . Para acceder a la información contenida en ese objeto, se procede igual que si se tratase de un array, recorriendo primero los datos de todas las ambulancias, luego los hospitales y finalmente los pacientes. Se creará un icono identificativo de cada agente vivo en el entorno en el mapa cargado con `GMarker` y `GIcon`, y éste se irá modificando o moviendo según los datos recibidos desde Java. En cada uno de esos iconos, se añade un *pop-up* de información que salta cada vez que el usuario mueve el cursor del ratón por encima de un marcador, evento recogido con la clase `GEvent`. Ese *pop-up*, que consiste en el resumen del estado del agente en forma de texto, se muestra con el método ***openInfoWindowHtml()*** de la clase `GMarker`.

3.6.3.3.2. Hilos de ejecución

Toda aplicación SWT necesita utilizar un objeto de la clase `Display`, el cual toma el valor de la pantalla en la que se creara la interfaz gráfica. Cuando se trabaja con SWT, hay que tener en cuenta que ese objeto `Display` va a crear un único hilo de ejecución, denominado "UI-Thread" (hilo visual). Todos los componentes y ventanas (instancias de la clase `Shell`) que formen parte de la interfaz se deberán crear en ese mismo hilo, que será el encargado de capturar todos los eventos que ocurran mediante el siguiente fragmento de código:

```
while (!shell.isDisposed ()) {
     if (!display.readAndDispatch ())
         display.sleep ();
}
display.dispose ();
```

Con este bucle, se mantiene la ventana a la escucha de que ocurra algún evento en alguno de sus componentes. Mientras no se detecte ningún evento, la ventana se queda en estado de espera. Sólo se saldrá de ese bucle cuando el usuario cierre la `Shell`, finalizando el `Display` y todos los componentes que contenga.

Al quedarse la ventana bloqueada esperando a que ocurran eventos, se impide que se ejecute otra parte de la aplicación a la vez que se muestra la interfaz. En nuestro proyecto, por ejemplo, no se podría ver el modulo *Visualizador* a la vez que están los agentes comunicándose mediante JADE. Por ello, es necesario que ambas partes se lancen en hilos diferentes y que puedan ejecutarse de forma concurrente.

El problema aparece cuando se quiere llamar a la interfaz desde alguna parte del código que se está ejecutando en otro hilo diferente al "UI-Thread". SWT no permite que se acceda a componentes gráficos fuera de ese hilo de la forma habitual, sino que se debe proveer un código que implemente la interfaz `Runnable` y que dentro de su método *`run()`* se llame a la parte del código del "UI-Thread" que se desee modificar. Ese `Runnable` se utilizará dentro de los métodos *`syncExec(Runnable)`* o *`asyncExec(Runnable)`* del objeto `Display`. Mediante cualquiera de los dos procedimientos se consigue modificar la interfaz gráfica desde otro hilo distinto al visual, con la diferencia de que el primero lo hace de forma síncrona, bloqueando el hilo no visual hasta que se complete el procedimiento pasado por argumentos, y el segundo de forma asíncrona, ejecutando lo que se pide sin bloquearse hasta su finalización.

3.6.3.4. Almacenamiento de datos

En este proyecto, el almacenamiento de información de forma persistente es fundamental. De esta forma, guardando los datos de una simulación, se puede posteriormente visualizarla con el subsistema *Visualizador*, sólo leyendo el fichero que contiene la información, sin necesidad de tener que usar el *Simulador*, encargado de ejecutar simulaciones. Pero no es la única parte de la aplicación que requiere ser salvada, ya que también las configuraciones que se crean en el *Setup* se deben poder guardar para poder repetir experimentos de simulación con los mismos parámetros de configuración, y así estudiar el comportamiento del entorno cuando se introduce aleatoriedad en sólo algunos de sus parámetros.

Estas dos operaciones, se corresponden con los casos de uso descritos en la sección de Análisis (ver diagrama 3.4), denominados *Save configuration* y *Save simulation*. Para su almacenamiento, se ha hecho uso de una librería Java que implementa la serialización de objetos en formato XML y viceversa, llamada *XStream*. Su simplicidad y facilidad de uso hacen que una tarea compleja como dar un formato XML a una clase Java se resuelva en una sola línea de código. XML (*Extensible Markup Language*)es un lenguaje para el intercambio de datos muy sencillo basado en etiquetas. Almacenar la información en este formato nos proporciona numerosas ventajas en cuanto al transporte de datos y la lectura de éstos.

En el primer caso, cuando se quiere almacenar en un fichero una configuración creada, basta con serializar el objeto `Configuration` entero, ya que ésta es la que recoge todos los parámetros especificados en el módulo *Setup*. En este caso, la utilización de la librería *XStream* es la siguiente:

```
FileWriter configFile =new FileWriter(configurationFich);
XStream xstream = new XStream();
ObjectOutputStream out = xstream.createObjectOutputStream(configFile);
out.writeObject(this);
out.close();
```

Como se puede observar, se instancia un objeto `XStream` y en él se crea un `ObjectOutputStream` para poder escribir datos en el fichero especificado. La escritura se encargará de dar el formato adecuado, transformando el objeto a la siguiente estructura XML:

- Se creará un único elemento raíz, llamado por defecto <object-stream>.
- Cada atributo de la clase se convertirá en una etiqueta XML con el mismo nombre del atributo, y su contenido será el valor que se le haya dado a dicho atributo. Para cambiar el nombre de la etiqueta, se puede usar la función *xstream.alias(<alias>, <nombre-atributo>);* , la cual se encargará de sustituir el nombre del atributo por el alias especificado.
- Al hacer el cierre del `ObjectOutputStream`, se cierra el elemento raíz.

En el segundo caso, cuando se quiere guardar una simulación que se está ejecutando, se requiere almacenar tanto el estado del entorno de cada paso de simulación, como la lista de acciones y mensajes que se pasan entre los agentes del entorno. Todos esos datos, junto al número de paso en el que estamos y su *timestep* correspondiente, se guardan en objetos de la clase *StepSimulation*. La idea inicial fue almacenar cada paso de simulación en una lista y serializar al final de la ejecución la lista entera, pero muchos de los objetos a serializar eran referencias a otros objetos, por lo que no se iban salvando los diferentes estados internos del entorno, si no que todos los estados acababan apuntando al último estado simulado. Por ello, se ha implementado un sistema en el que el fichero en el cual se van a escribir todos los pasos de simulación se mantiene abierto durante toda la simulación y al ejecutar cada paso, se manda escribir el nuevo *StepSimulation* instanciado. La regla de oro de cualquier fichero XML es que sólo puede tener un elemento raíz, por lo que, si usamos el mismo procedimiento que se usa en la configuración, se mantiene una única raíz que se compondrá de tantos subelementos de tipo *StepSimulation* como se generen. Para que en la visualización offline se sepa cuantos pasos de simulación existen en el fichero, se crea una primera etiqueta XML que contiene el número total de *steps* que se han simulado y el *timestep* asociado.

La lectura de los ficheros XML es tan sencilla como su escritura. En vez de la creación de un `ObjectOutputStream` en el objeto `XStream`, se crea un `ObjectInputStream` que permita leer datos del fichero especificado mediante la función *`readObject()`*. Esta función leerá objetos XML que se encuentran dentro de la raíz del documento, y después se podrá trabajar con ellos en el código Java de la forma habitual. Si en vez de un objeto se sabe que se va a leer un tipo primitivo de Java, se tienen funciones de lectura adecuadas a cada tipo, por ejemplo, *`readInt()`* para leer números enteros.

A parte de estas dos funcionalidades de lectura y escritura de documentos XML, existen otras partes del sistema que van a utilizar esta librería. Una de ellas es en el módulo *Setup*, cuando el usuario puede subir datos de ambulancias u hospitales desde documentos XML, sin necesidad de ir configurando una a una cada ambulancia u hospital. La otra parte, es la encargada de enviar la información de la simulación al mapa del subsistema *Visualizador*, información que se envía en formato JSon, basado de JavaScript. A continuación, se explican con más detalle estos tres tipos nuevos de almacenamiento de datos:

- Ambulancias y Hospitales cargados desde fichero: cada ambulancia del sistema debe tener configurados una serie de parámetros que la definen en el entorno. Estos parámetros se recogen en la clase del subsistema *Simulador* `tAmbulanceParameters`. Cuando una ambulancia se crea mediante el módulo *Setup*, se solicitan al usuario todos esos datos y se almacena en la configuración de la simulación una lista con todas las ambulancias creadas y su `tAmbulanceParameters` asociado. Si se desea cargar un fichero con datos de ambulancias, ese fichero debe contener un documento XML en su interior que cumpla una cierta estructura. Esta estructura viene definida por la siguiente DTD[1]:

  ```
  <!ELEMENT object-stream (list)>
  <!ELEMENT list (es.urjc.ia.at.mhealth.simulator.setup.configuration.tAmbulanceParameters)+ >
  <!ELEMENT es.urjc.ia.at.mhealth.simulator.setup.configuration.tAmbulanceParameters (alias,
  ```

[1]Una DTD (Document Type Definition) sirve para describir el formato de datos que va a tener un documento XML, sus restricciones y la sintaxis de éste.

```
route, levelCapacity )>

<!ELEMENT alias (#PCDATA)>

<!ELEMENT route (string)+ >

<!ELEMENT string (#PCDATA)>

<!ELEMENT levelCapacity (maxLevels, level)>

<!ELEMENT maxLevels (#PCDATA)>

<!ELEMENT level (int)+ >

<!ELEMENT int (#PCDATA)>
```

Un ejemplo de documento XML de ambulancias válido sería:

```
<object-stream>
    <list>
        <es.urjc.ia.at.mhealth.simulator.setup.configuration.tAmbulanceParameters>
            <alias>ambulance1</alias>
            <route>
                <string>A: 40.352618,-3.762115</string>
                <string>B: 40.362121,-3.771715</string>
            </route>
            <levelCapacity>
                <maxLevels>2</maxLevels>
                <level>
                    <int>20</int>
                    <int>40</int>
                </level>
            </levelCapacity>
        </es.urjc.ia.at.mhealth.simulator.setup.configuration.tAmbulanceParameters>
    </list>
</object-stream>
```

Si se usa la librería *XStream* para leer el contenido de un fichero de ambulancias, se podrá leer sin problema una lista Java que contiene un conjunto de objetos `tAmbulanceParameters`, ya que la correspondencia entre la estructura del documento XML y la clase indicada es válida, debido a que las etiquetas que se usan

en el documento XML tienen los nombres de los atributos de la clase que define los parámetros de las ambulancias.

En el caso de los hospitales, el procedimiento es el mismo, pero cada hospital se define mediante una instancia de la clase Java `tHospitalParameters`. El fichero con el documento XML que se quiera leer debe contener igualmente una lista Java, pero esa lista almacenará objetos válidos para los hospitales, con sus etiquetas correspondientes. Por ello, la estructura del documento XML se validará con la siguiente DTD:

```
 <!ELEMENT object-stream (list)>
<!ELEMENT list (es.urjc.ia.at.mhealth.simulator.setup.configuration.tHospitalParameters)+ >
<!ELEMENT es.urjc.ia.at.mhealth.simulator.setup.configuration.tHospitalParameters (alias, level-
Urgency, numBeds, numMembers, location )>
<!ELEMENT alias (#PCDATA)>
<!ELEMENT levelUrgency (#PCDATA)>
<!ELEMENT numBeds (#PCDATA)>
<!ELEMENT numMembers (#PCDATA)>
<!ELEMENT location (#PCDATA)>
```

Un ejemplo de documento XML de hospitales válido sería:

```
<object-stream>
    <list>
        <es.urjc.ia.at.mhealth.simulator.setup.configuration.tHospitalParameters>
            <alias>Hospital Carlos III</alias>
            <levelUrgency>1</levelUrgency>
            <numBeds>98</numBeds>
            <numMembers>20</numMembers>
            <location>
                Calle de Sinesio Delgado, 12-16, 28029 Madrid, Espanya (40.430224,-3.680420)
            </location>
        </es.urjc.ia.at.mhealth.simulator.setup.configuration.tHospitalParameters>
        <es.urjc.ia.at.mhealth.simulator.setup.configuration.tHospitalParameters>
            <alias>H. Infantil Universitario Ninyo Jesus</alias>
```

```
            <levelUrgency>1</levelUrgency>
            <numBeds>199</numBeds>
            <numMembers>20</numMembers>
            <location>
                Calle de Pio Baroja, 28009 Madrid, Espanya (40.414322,-3.676634)
            </location>
        </es.urjc.ia.at.mhealth.simulator.setup.configuration.tHospitalParameters>
    </list>
</object-stream>
```

- Intercambio de datos con el mapa del *Visualizador*: JSON es la abreviatura de *Javascript Object Notation* y se trata de una forma de almacenar información de forma organizada y de fácil acceso. Tal y como indica su nombre, es una notación basada en JavaScript, por lo que su uso dentro de métodos JavaScript simplifica la tarea del intercambio de información entre Java y este lenguaje.

  Haciendo uso de la misma librería *XStream*, se pueden serializar objetos en formato JSON inicializando el objeto `XStream` con el *driver JSON* apropiado. En este caso, se ha usado la librería auxiliar *JettisonMappedXmlDriver*, la cual nos proporciona las funciones necesarias para convertir los objetos que necesitamos pasar al mapa del *Visualizador* en notación JSON. Este proceso se realiza de la siguiente forma:

```
JSONStep jsonStep = new JSONStep();
XStream xstream = new XStream(new JettisonMappedXmlDriver());
xstream.alias(''data'', JSONStep.class);
xstream.alias(''patients'', JSONPatient.class);
xstream.alias(''hospitals'', JSONHospital.class);
xstream.alias(''ambulances'', JSONAmbulance.class);
String JsonString = xstream.toXML(jsonStep);
```

  Con este fragmento de código se consiguen serializar los objetos que almacenan los datos de pacientes, hospitales y ambulancias que queremos visualizar en el mapa.

La clase `JSONStep` contiene todos esos datos en forma de objetos `JSONPatient`, `JSONHospital` y `JSONAmbulance`. A cada tipo de objeto se le ha dado un alias para facilitar su uso. Finalmente, todo el objeto se ha serializado pero esta vez, en vez de guardarse en un fichero, se ha guardado en una cadena de texto mediante la función *`toXML()`*. Esta función, serializaría en formato XML si no se especificase ningún *driver*, pero al indicarse que se quiere usar *JettisonMappedXmlDriver*, se serializa en formato JSON.

Esta cadena de texto se enviará al mapa del *Visualizador*, a través de la función que nos proporciona el navegador de SWT para ejecutar una función JavaScript. En este caso, la función será:

```
browser.execute(''drawSimulationStep(''+JsonString+'')'')
```

Dentro del fichero HTML que está cargado en el navegador, se encuentra el método JavaScript *`drawSimulationStep()`*. En este método, se procesa la cadena *JsonString* que ha sido serializada, transformándola a un objeto JavaScript mediante la función *`eval()`* de la siguiente forma:

```
var step = eval( ''('' + jsonString + '')'' );
```

`Step` será un objeto con el que se pueda trabajar, accediendo a cada atributo que contenga de la forma habitual en JavaScript.

3.6.4. Herramientas

Los programas y herramientas utilizados para el desarrollo del proyecto han sido los siguientes:

- Eclipse: se trata de un entorno de desarrollo integrado *open-source* que puede ser mejorado mediante plugins. La versión utilizada a sido "Galileo", disponible en su página Web oficial [**?**].

- Jude: para realizar toda la descripción informática del proyecto, se ha usado la herramienta *Jude-community*. Este entorno permite la creación de diversos diagramas descriptivos de programación orientada a objetos usando el lenguaje de modelado UML. El ejecutable puede descargarse desde su Web oficial [**?**].

- Jade (*Java Agent DEvelopment Framework*): es un framework *open-source* para el desarrollo de agentes software en Java, se trata de un *middle-ware* que cumple con las especificaciones *FIPA* y proporciona un conjunto de herramientas visuales que facilitan el desarrollo y la depuración de sistemas multi-agente. La versión utilizada ha sido la 3.7, disponible en su página Web oficial [**?**].

- Protégé: es un editor de ontologías *open-source* el cual ha sido utilizado para la edición de la ontología de la aplicación. La versión utilizada ha sido la 3.4.2, disponible en su página Web oficial [**?**]. Ademas a esta herramienta le hemos incluido un plugin llamado *OntologyBeanGenerator* el cual genera archivos java representando la ontología de tal manera que pueda ser usada con *Jade*.

- JDK: es un software que provee herramientas de desarrollo para la creación de programas en java, se ha utilizado la versión 6, disponible en su página Web oficial [**?**]

- SWT (*Standard Widget Toolkit*): es un framework open-source para construir interfaces gráficas en Java, basado en la utilización de un conjunto de componentes

visuales, llamados *widgets*. Se ha usado SWT en su versión 3.5, disponible en su página Web oficial [**?**].

SWT no provee un entorno de desarrollo visual para el diseño de las interfaces gráficas, por lo que se ha usado, junto al IDE Eclipse, un plugin llamado "Visual Editor". Este plug-in permite crear clases visuales e integrar en ellas widgets de SWT de forma muy sencilla, sin necesidad de tener conocimientos avanzados sobre programación de interfaces gráficas en Java. Su versión 1.4.0 está disponible en [**?**].

- Xstream: esta librería Java se ha utilizado para serializar y deserializar objetos a formato XML y a formato JSON. La versión usada ha sido la 1.3.1, disponible en [**?**]. Para exportar a JSON, ha sido necesaria una librería auxiliar que serializa los objetos a ese formato llamada *JettisonMappedXmlDriver* (versión 1.0.1).

- Subclipse: se trata de un plug-in para Eclipse que proporciona soporte a Subversion (software *open-source* de sistema de control de versiones). La versión que se ha utilizado ha sido la 1.6, la cual se puede encontrar en su página Web oficial [**?**].

- API de JavaScript de GoogleMaps: existen diversas APIs de Google Maps para insertar mapas en páginas Web. La que se ha usado en el proyecto es la API basada en JavaScript, versión 2. Su utilización es muy sencilla y permite personalizar los mapas manipulándolos y añadiéndoles contenido. Toda la documentación necesaria para el desarrollo de mapas con Google Maps se encuentra en la página [**?**]. El uso que se ha hecho de esta API para el proyecto *mHealth* se explica con más detalle en el apartado 3.6.3.3.1 de esta sección de Implementación.

- Texmaker: para la elaboración de esta memoria se ha usado este editor de LaTeXque integra todas las herramientas necesarias para desarrollar documentos.

3.7. Evaluación

El objetivo de esta fase de desarrollo es conseguir la detección de posibles errores de implementación y ver que el sistema es válido frente a los requisitos especificados al principio.

3.7.1. Demostración del funcionamiento

En esta sección, se presenta una demostración del sistema implementado con la ejecución de un escenario concreto. En ella, se muestra cómo configurar una simulación y cómo ejecutarla, para ver los resultados de la interacción entre los agentes del escenario creado.

3.7.1.1. Descripción del escenario

Para la demostración vamos a desarrollar un escenario de emergencias médicas compuesto de cuatro ambulancias coordinadas a través de un único centro de emergencias, y dos hospitales con diferentes niveles de urgencia. En total, aparecerán 10 pacientes a lo largo de la simulación del escenario, con diferentes niveles de enfermedad y en distintas posiciones del territorio. Con esta simulación, se verá como el centro de emergencias es capaz de coordinar los recursos de las que dispone para conseguir atender con éxito a todos los pacientes que soliciten asistencia.

3.7.1.2. Configuración del escenario

3.7.1.2.1. Configuración de parámetros de simulación

El primer paso es definir los parámetros del escenario de la simulación, así como definir la cantidad de actores que participarán en el entorno. Para ello, hay que lanzar la aplicación *mHealth* y en su menú inicial, elegir la opción "*Create a new configuration to simulate*". Una vez accedemos a la configuración del escenario, nos aparece

la ventana de la figura 3.60 en la que indicaremos sus parámetros generales. En el panel *Simulation files* introducimos en el campo *Simulation identifier* el identificador "demo_evaluacion" como nombre de los ficheros de configuración y simulación que se crearán en los directorios indicados en los campos *Configuration file path* y *Historical file path* respectivamente. Mediante el panel *Time and Steps* indicamos 200 *steps* de duración de la simulación en la que la duración de cada *step* será de 970 milisegundos a nivel de ejecución equivalentes a 15 minutos reales. Por último, seleccionamos en el mapa una zona en la que se desarrollará el escenario, confirmando la selección mediante el botón *Finish map* del panel *Environment's Map*. Tras ello, pulsamos el botón *Next* para seguir con los demás pasos de la configuración.

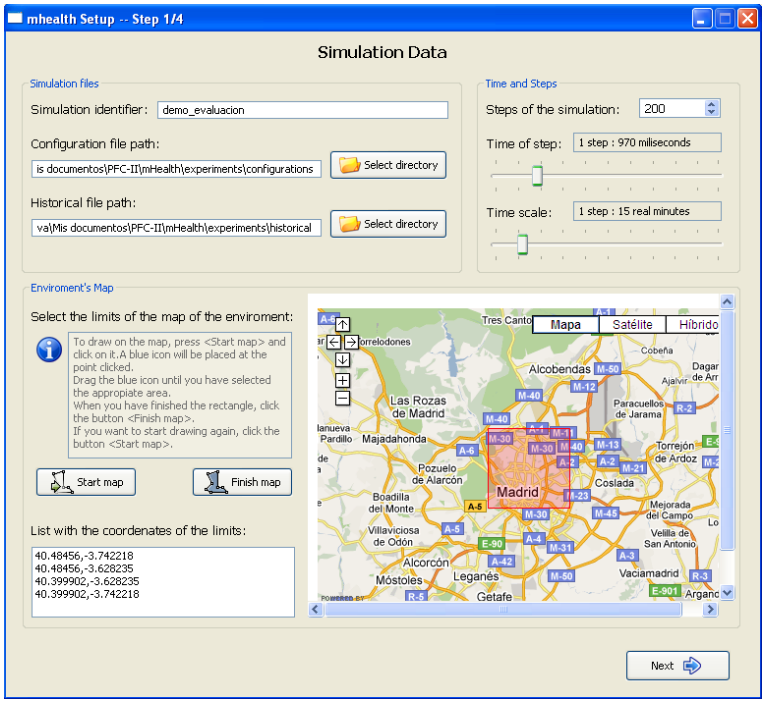


Figura 3.60: Configuración de parámetros de simulación

3.7.1.2.2. Configuración de centros de emergencia, pacientes y distribuciones

La siguiente ventana que aparece es la relativa a la configuración de los centros de emergencia, pacientes y distribuciones, ver 3.61. En el panel *List of Emergency Center* se encuentra un centro de emergencia llamado "SUMMA" que hemos creado con 5 operadores. En el panel *Patient Parameters* indicamos que el número total de pacientes que habrá durante la simulación será 10 y su nivel máximo de enfermedad será 5. Por último, seleccionamos las distintas distribuciones que dotarán de aleatoriedad a la simulación. La explicación de cada distribución implementada se puede ver en el apartado 3.6.3.2.2 de la sección de Implementación. En este escenario, elegimos las siguientes distribuciones: los pacientes se generarán de uno en uno por cada *step*, se crea mayor densidad de población en el centro del área de simulación, el nivel de enfermedad de los pacientes se generará de manera exponencial y la ocupación de los hospitales será uniforme.

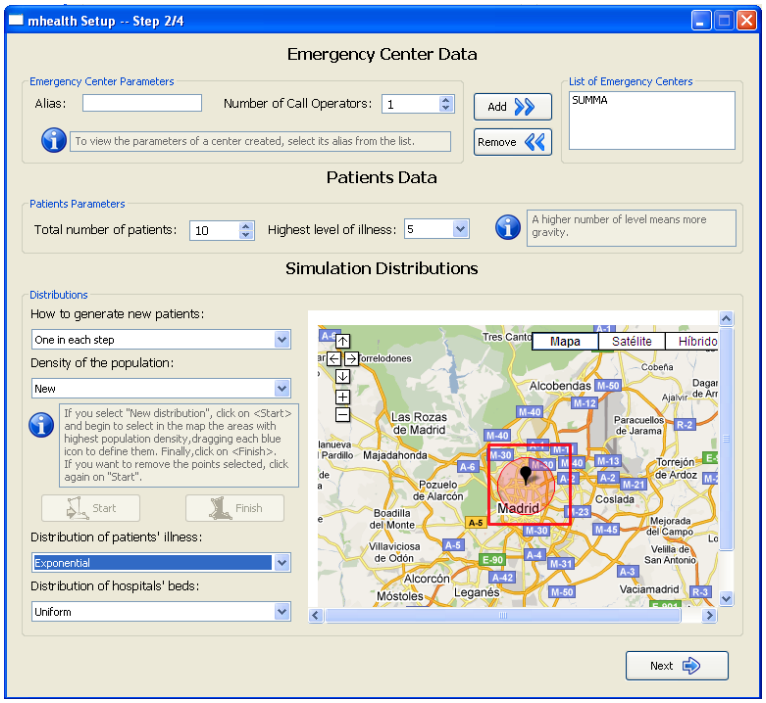


Figura 3.61: Configuración de centros de emergencia, pacientes y distribuciones

3.7.1.2.3. Configuración de ambulancias

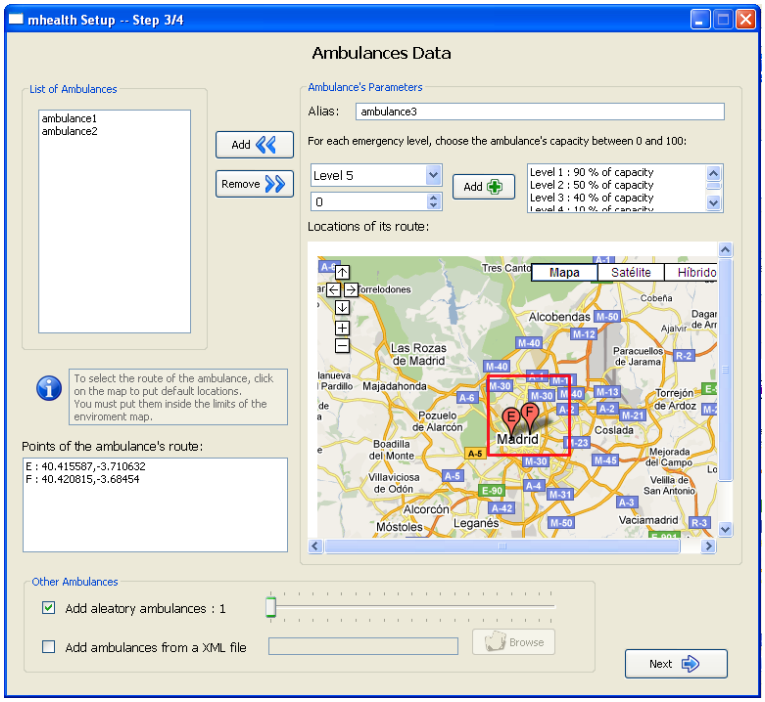


Figura 3.62: Configuración de ambulancias

Después de configurar los centros de emergencia, pacientes y distribuciones, tenemos que configurar la información relativa a las ambulancias. Como podemos observar en el panel *List of Ambulances* de la figura 3.62 hemos creado manualmente dos ambulancias y además estamos creando una nueva ambulancia que se llamará "ambulance3", se moverá entre los marcadores "E" y "F" del mapa y se le asignaran los siguientes porcentajes de adecuación a los distintos niveles de enfermedad (la explicación de este concepto se puede ver en la descripción del estado interno de una ambulancia, en el apartado 3.6.3.2.1 de la sección de Implementación): nivel 1 con un 90 % de eficiencia, nivel 2 con un 50 %, nivel 3 con un 40 %, nivel 4 con un 10 % y nivel 5 con un 0 %. Además crearemos una ambulancia aleatoria a través del panel *Other ambulances*.

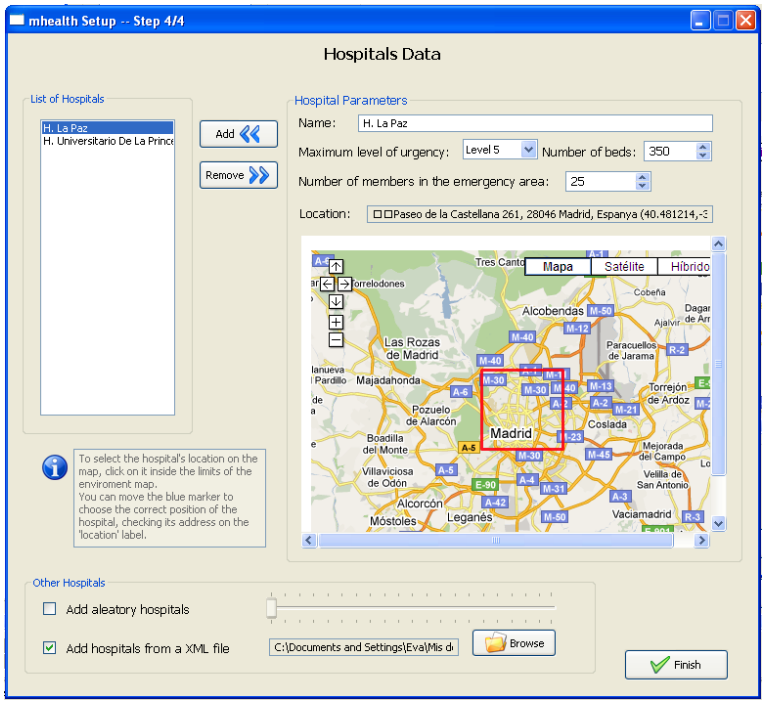


Figura 3.63: Configuración de hospitales

3.7.1.2.4. Configuración de hospitales

Por último hay que configurar la información relativa a los hospitales, ver figura 3.63. Mediante la opción de añadir hospitales a partir de un fichero XML situada en el panel *Other Hospitals* hemos incluido dos hospitales a la simulación: "H. La Paz" y "H. Universitario De La Princesa". Los datos con los que se ha configurado el "H. La paz" los podemos observar en la captura y son los siguientes: el hospital es capaz de atender como máximo a pacientes con un nivel de enfermedad 5, tiene un total de 350 camas, 25 empleados dedicados a las acciones relativas a la simulación y se encuentra en el Paseo de la Castellana 261 en Madrid. Y el hospital "H. Universitario De La Princesa" es capaz de atender como máximo a pacientes con un nivel de enfermedad 3, tiene un total de 515 camas, 10 empleados dedicados a las acciones relativas a la simulación y se encuentra en el Calle de Diego de León 62 en Madrid.

3.7.1.3. Ejecutar la simulación del escenario configurado

Una vez está configurado el escenario, se procede a ejecutar la simulación. En este ejemplo, como el escenario se ha configurado en el momento, el sistema pasará a su simulación directamente. Para poder observar lo que va pasando entre los agentes del entorno gráficamente, seleccionamos la opción "*Show the simulation with a graphic interface*" para usar el visualizador implementado.

En ese momento, se inicializa el contenedor de agentes JADE y aparecen los iconos de las ambulancias y hospitales que van a estar en el escenario simulado en el mapa de la interfaz. A la derecha, en el panel de información, se ve cómo comienzan a crearse los primeros pacientes, apareciendo sus nombres en el listado de pacientes uno a uno. Esto se debe a que se seleccionó la distribución "*One in each step*" para la generación de pacientes en la configuración inicial. En el mapa, se van colocando los iconos representativos de los pacientes en sus localizaciones correspondientes. Se puede observar cómo todos los iconos aparecen repartidos por el centro del mapa, debido a que la distribución de densidad de población seleccionada, agrupaba la población en un círculo centrado en los límites del mapa del entorno.

El centro coordinador SUMMA comienza a recibir las primeras llamadas de los pacientes enfermos. El centro, como máximo, podrá atender 5 llamadas a la vez pero en este escenario, nunca llega a saturarse la central de emergencias por ir apareciendo los pacientes de uno en uno y no de golpe, ya que si aparece más de un paciente en un mismo *timestep*, llamarían a la vez al centro, pudiendo llegar a ocupar a todos sus operadores si son más de 5 pacientes.

En el panel informativo de los centros coordinadores, se muestra en cada momento el número de misiones activas que está gestionando el SUMMA. Como máximo, estas misiones serán 4, debido a que sólo existen 4 ambulancias en el escenario. Si analizamos toda la secuencia de llamadas recibidas por el SUMMA y las misiones solicitadas para cada llamada, dentro del histórico de acciones que aparece cuando se selecciona su nombre en la interfaz, se puede ver cómo el SUMMA almacena una cola

de pacientes pendientes de asignarles ambulancias, ya que en cuanto una ambulancia se queda disponible, escoge a uno de los pacientes en cola y le solicita a la ambulancia que vaya a asistirle:

```
2-4: Skip

4: Tell the pat0 to wait for an ambulance

5: Skip

5: Tell the pat1 to wait for an ambulance

5: Mission assignement for patient pat0

6: Skip

6: Tell the pat2 to wait for an ambulance

6: Mission assignement for patient pat1

7: Skip

7: Mission assignement for patient pat2

7: Tell the pat3 to wait for an ambulance

8: Skip

8: Mission assignement for patient pat3

8: Tell the pat4 to wait for an ambulance

9: Skip

9: Tell the pat5 to wait for an ambulance

10: Skip

10: Tell the pat6 to wait for an ambulance

11: Skip

11: Tell the pat7 to wait for an ambulance

12: Skip

12: Tell the pat8 to wait for an ambulance

13: Skip

13: Tell the pat9 to wait for an ambulance

14-29: Skip

29: Mission assignement for patient pat7
```

```
30-56: Skip

56: Mission assignement for patient pat6

57-77: Skip

77: Mission assignement for patient pat8

78: Skip

78: Mission assignement for patient pat9

79-81: Skip

81: Mission assignement for patient pat4

82-115: Skip

115: Mission assignement for patient pat5

116-200: Skip
```

Los números que aparecen al comienzo de cada acción, se corresponden con el paso de simulación en el que ha tenido lugar dicha acción. Si revisamos el histórico de acciones de cada paciente, se puede ver cómo llaman al centro coordinador justo en el *timestep* anterior a la respuesta. Como ejemplo, vemos la información del paciente 0 en el 3.64: el paciente aparece en el sistema en el segundo paso de simulación, ya que el primero es cuando se lanzan las ambulancias, hospitales y centros. En el tercer *timestep*, llama al SUMMA y en el cuarto, el SUMMA responde al paciente.

Cuando el SUMMA asigna una misión a una ambulancia y ésta la acepta, la ambulancia comienza a moverse hacia el paciente asignado. A modo de ejemplo, vamos a ver tres casos posibles en la asistencia de una ambulancia a un paciente:

- El paciente se cura con la asistencia de la ambulancia: a la ambulancia *ambulance3* se le asigna la misión del paciente *Pat5*. Este paciente, que llamó al centro coordinador en el *timestep 8*, no recibe la asistencia de la ambulancia hasta el *timestep 126*. Al tener el paciente un nivel de enfermedad leve (nivel 1) y la ambulancia poseer una capacidad del 90 % para ese nivel, en dos *timesteps* consigue curarle con éxito. Tras ello, la ambulancia cierra misión indicando al centro SUMMA que el paciente se curó satisfactoriamente. En la captura 3.65 se puede ver el momento justo en el que la ambulancia cierra misión y cómo se queda el

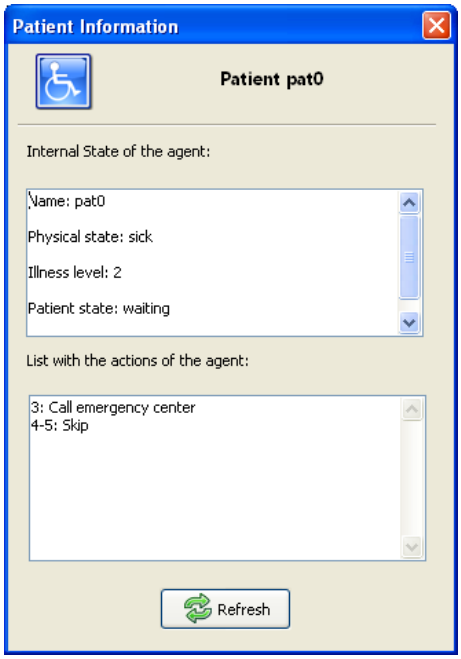


Figura 3.64: Información individual de un paciente

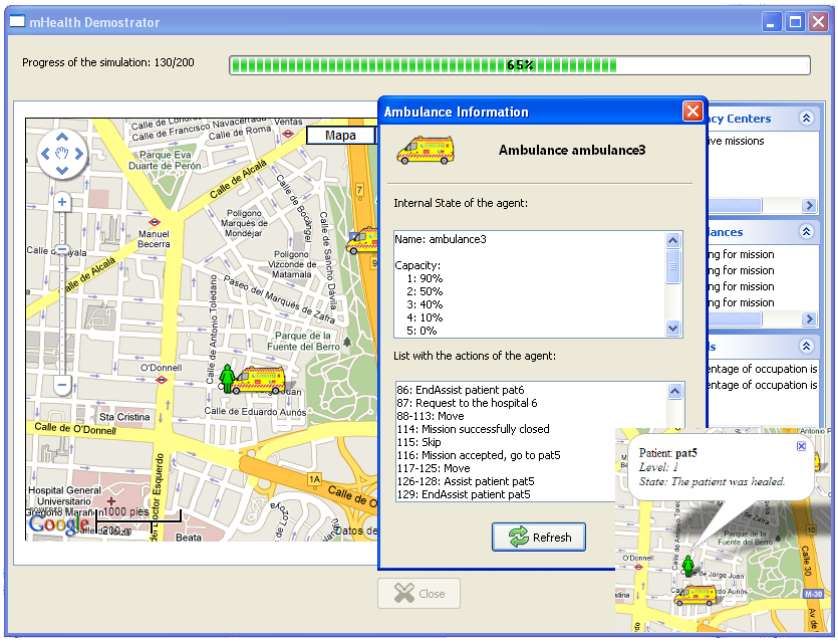


Figura 3.65: El paciente Pat5 se cura con la asistencia de la ambulancia 3

paciente de color verde (icono representativo de su curación) en el mapa mientras la ambulancia se aleja.

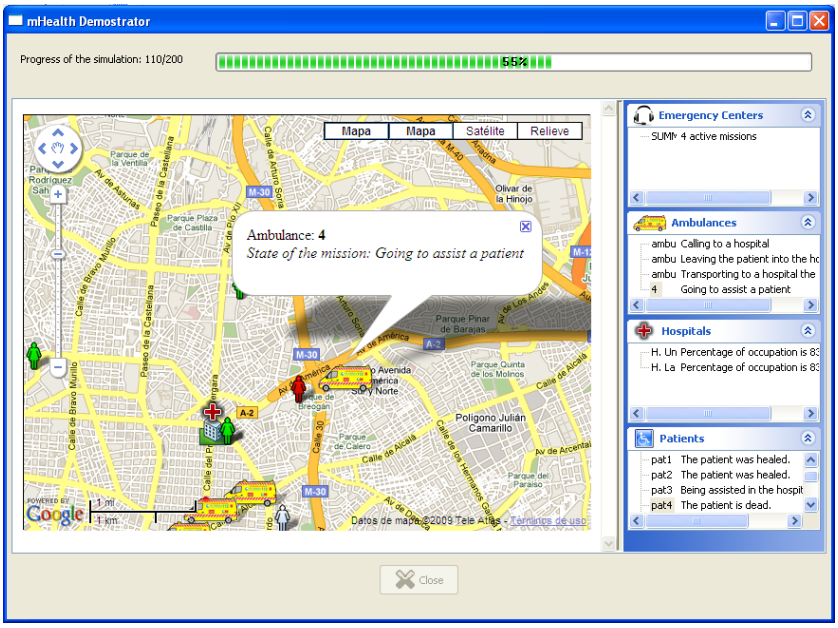


Figura 3.66: El paciente fallece antes de recibir asistencia médica

- Cuando llega la ambulancia el paciente ya ha muerto: el paciente *Pat4* se crea en el entorno con un nivel de enfermedad 5, el nivel más grave. Nada más aparecer, el paciente llama al SUMMA pero al estar todas las ambulancias ocupadas, debe esperar hasta el *timestep 81* en el que el SUMMA solicita a la ambulancia *4* (la que se creó aleatoriamente) que le vaya a asistir. En ese momento, la ambulancia comienza a desplazarse hasta el paciente pero a medio camino, *pat4* fallece, tal y como se puede ver en la figura 3.66. El icono del paciente se vuelve rojo pero la ambulancia hasta que no esta con el paciente, no es consciente de su muerte. En el histórico de acciones de la ambulancia *4* se puede ver cómo cierra misión en estado fallido:

```
82: Mission accepted, go to pat4

83-114: Move

115: Assist patient pat4
```

```
116: EndAssist patient pat4

117: Close mission with failed state
```

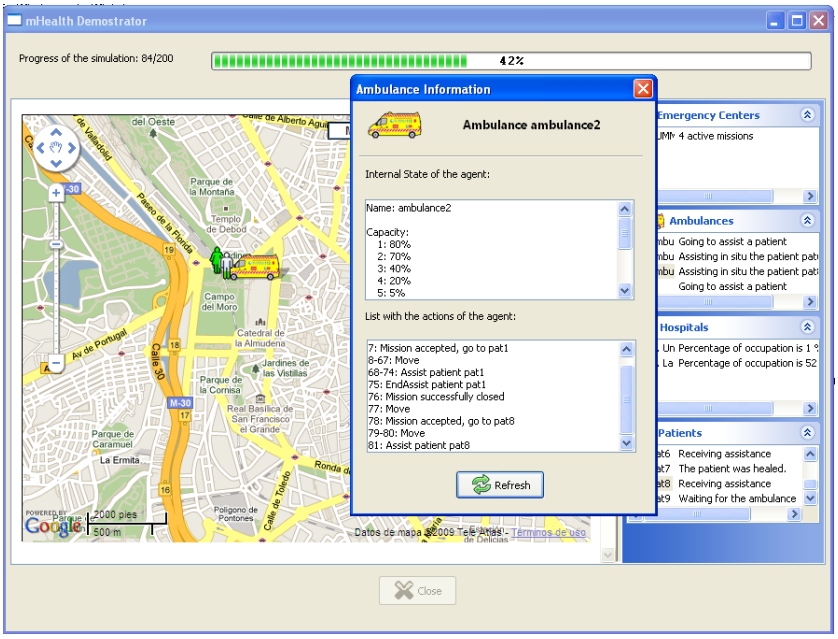


Figura 3.67: El paciente continúa enfermo después de la asistencia in situ

- El paciente no se cura con la asistencia de la ambulancia, así que se le traslada a un hospital: la ambulancia *ambulance2* se dirige a asistir al paciente *pat8*, con nivel de enfermedad 3. Para ese nivel, tiene una capacidad del 40 %, por lo que no consigue curarle *in situ* tras 10 *timesteps* de asistencia (la asistencia *in situ* se puede ver en la figura 3.67). Es entonces cuando decide llevarle a un hospital, captura 3.68, para que le ingresen y puedan curarle.

Este último caso, implica la interacción de las ambulancias con los hospitales del entorno, de forma que las ambulancias pregunten a los hospitales si hay disponibilidad para ingresar a un paciente en concreto. Una vez que un hospital acepta el envío de un paciente, éste le admitirá cuando llegue desde la ambulancia que llamó. En nuestra simulación se aprecia lo que sucede cuando una ambulancia traslada a un paciente enfermo, y cuando traslada a un paciente que fallece durante ese desplazamiento:

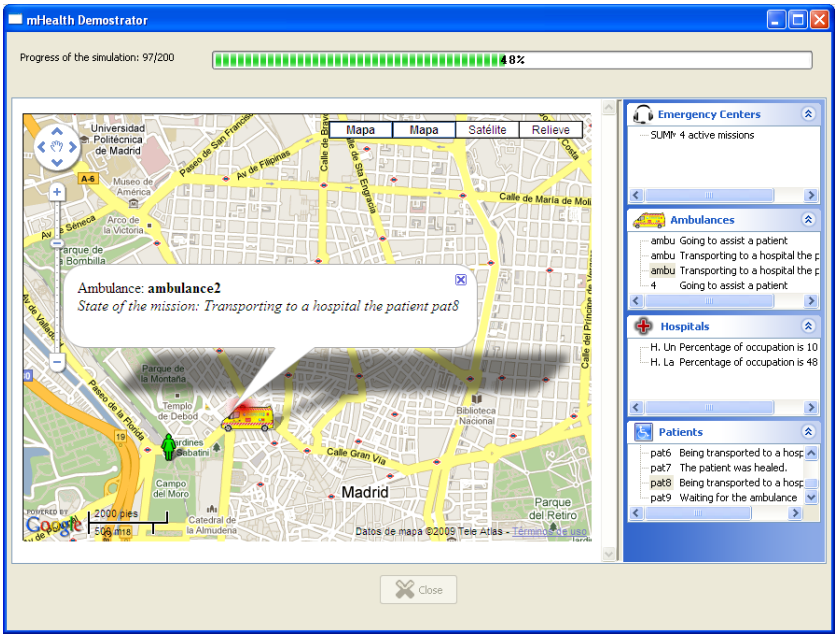


Figura 3.68: El paciente es trasladado a un hospital

- El paciente llega enfermo al hospital y éste le cura: continuando con el caso anterior, en el que el paciente *Pat8* era asistido por la ambulancia *ambulance2*, se ve como ésta, al no poder curar al paciente, decide trasladarle a un hospital. *Ambulance2* llama al hospital 6 (número asociado al hospital universitario La Princesa), y éste acepta el envío del paciente. La comunicación entre ambos se puede ver en el siguiente resumen de sus históricos de acciones:

```
Ambulancia, 93: Request to the hospital 6

Hospital, 94: Answer to a request of ambulance 2 with true

Ambulancia, 94-119: Move

Hospital, 119: Admit patient pat8 from ambulance 2

Ambulancia, 120: Mission successfully closed

Hospital, 120-133: Skip

Hospital, 134: Release patient pat8
```

  La ambulancia, al dejar al paciente en el hospital, cierra la misión con éxito ya que, aunque no ha podido curarle, le ha trasladado a un hospital para que siga

con asistencia médica. Finalmente, el paciente es dado de alta con éxito, ya que se le ha conseguido curar tras 13 *timesteps* de asistencia hospitalaria.

- El paciente fallece en el traslado hacia el hospital: el paciente *Pat9*, con nivel de enfermedad 5, recibe la asistencia *in situ* de la ambulancia *ambulance1*. Ésta, después de 3 *timesteps* de asistencia, decide trasladarle a un hospital por su gravedad, ya que la ambulancia sólo tiene un 25 % de capacidad para ese nivel de enfermedad. Llama al hospital La Princesa, el cual tiene un nivel de urgencias 3 (inferior al del paciente) y éste acepta el traslado de *Pat9* hasta sus instalaciones. El problema es que justo a medio camino, el paciente fallece dentro de la ambulancia. Aún así, la ambulancia debe llevar al paciente al hospital asignado, dejarle allí y cerrar misión con éxito, a pesar de que el paciente haya fallecido, pero como en cuanto deja de atender al paciente su misión es trasladarlo a un hospital, ese objetivo si lo ha cumplido. El hospital admite al paciente en urgencias y acto seguido le libera, al darse cuenta de que ha fallecido. La secuencia de acciones entre hospital y ambulancia es la siguiente:

```
 Ambulancia, 110: Request to the hospital 6
Hospital, 111: Answer to a request of ambulance 1 with true
Ambulancia, 111-126: Move
Hospital, 117: Admit patient pat9 from ambulance 1
Ambulancia, 117: Mission successfully closed
Hospital, 118: Release patient pat9
```

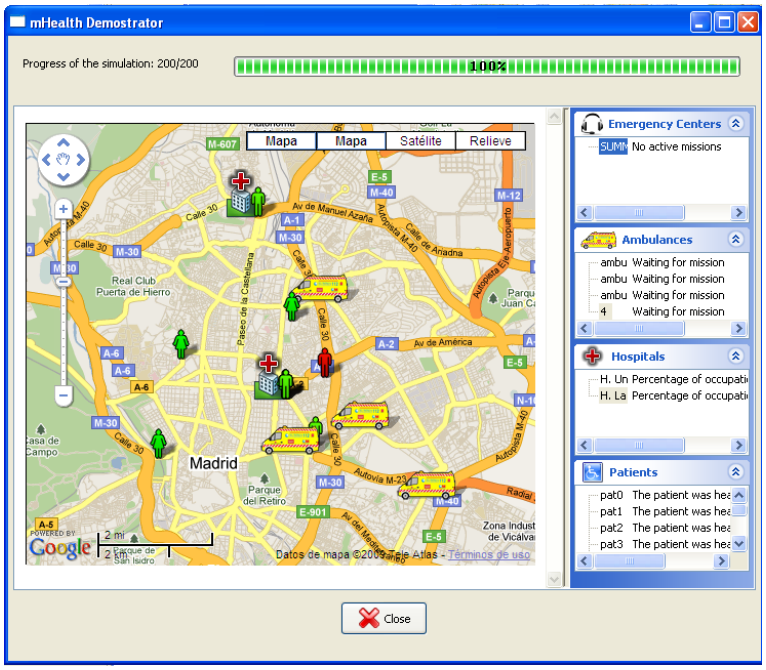


Figura 3.69: Pantalla final de la simulación

Al final de la simulación, la pantalla finaliza con la visión de todo lo ocurrido en el último *timestep*, tal y como se muestra en la figura 3.69. En resumen, de los 10 pacientes que han ido apareciendo a lo largo de la simulación, se ha conseguido curar a 8 mientras que 2 (ambos con el nivel de gravedad máximo) han fallecido.

Para finalizar el programa, se debe cerrar la ventana del visualizador con el botón *Close* para poder volver al menú inicial, y ahí presionar el botón *Cancel* para salir del sistema.

3.7.1.4. Visualización del histórico

La simulación del escenario anterior ha quedado almacenada en el fichero *historical_demo_evaluacion.xml*, el cual se adjunta en el CD-ROM para poder observar visualmente toda la explicación proporcionada en esta sección de evaluación.

Para ello, se debe ejecutar el programa *mHealth* y, en su menú principal, elegir la opción "*Load a simulation that already exists to view it*". El sistema nos pedirá la ruta

del fichero con el histórico a cargar. Para cargar la simulación anterior, se debe buscar dentro del CD-ROM la carpeta *mHealth/experiments/historical* el fichero *historical_demo_evaluacion.xml*. La configuración asociada, se encuentra en *mHealth/experiments-/configurations*, en el fichero XML *conf_demo_evaluacion.xml*.

Capítulo 4 Conclusiones

La principal contribución del proyecto ha sido el desarrollo de una primera versión del demostrador *mHealth*, capaz de recrear escenarios realistas del dominio de las emergencias médicas, el cual nos permite evaluar y probar distintas técnicas, métodos y algoritmos que se obtengan en el ámbito del proyecto *Agreement Technologies* (AT en adelante).

Como consecuencia de dicho logro, se han conseguido las siguientes metas:

- Se ha diseñado una arquitectura para el demostrador *mHealth* compuesta de tres subsistemas independientes: aplicación, simulador del entorno y visualizador de la simulación. Esta arquitectura viene dada por las tres grandes funcionalidades recogidas en los requisitos del sistema y presenta la ventaja de poder desarrollar de manera individual cada subsistema, para poder tener en un futuro distintas implementaciones de cada uno de ellos. Por ejemplo, podrían combinarse una aplicación de agentes basada en incentivos, con el entorno actual y con una nueva tecnología de visualización. Algunas líneas futuras del demostrador se explican con más detalles en el apartado 4.1.

- La aplicación desarrollada permite recrear de manera bastante realista todos los participantes involucrados en una emergencia médica del mundo real. Se ha

utilizado un paradigma basado en agentes para modelar cada actor, definiendo las acciones que son capaces de realizar en este dominio y la ontología empleada por todos ellos.

- Dado que utilizar un escenario real es prácticamente imposible (recursos, infraestructuras, vehículos, personas, etc), para dotar de mayor realismo al demostrador del proyecto se ha desarrollado un simulador de entornos físicos dentro del dominio de las emergencias médicas. Este simulador, permite representar dos conceptos imprescindibles en ese dominio: el tiempo y el espacio. De esta manera, podemos recrear el tiempo invertido en cada emergencia médica y ver la situación física de cada entidad participante en el entorno. Dicho entorno, representa todo aquello que rodea a los agentes y es el encargado de simular todas las acciones solicitadas por dichos agentes relacionadas con el mundo físico: llamadas telefónicas (comunicación entre agentes intercambiando mensajes entre ellos), movimientos, asistencias médicas, etc.

  Además, el simulador se compone de un módulo denominado *Setup*, que permite al usuario crear distintas configuraciones de escenarios de emergencias médicas, ya que el objetivo final del proyecto es la experimentación de distintos escenarios para poder evaluar los diversos mecanismos y modelos obtenidos en el proyecto AT.

- Finalmente, para poder evaluar de manera más eficiente toda la actividad desarrollada en cada simulación, se ha realizado un visualizador gráfico que, por medio de mapas y una interfaz gráfica, permite al usuario comprender mejor lo que está pasando en el entorno, viendo la interacción entre los actores así como el estado de dicho entorno en cada momento. Además, esta interfaz permite obtener mayor nivel de detalle seleccionando solo la información asociada a un agente concreto y mostrarla de forma individual, para entender mejor el comportamiento del agente visualizado.

En cuanto a las dificultades presentadas durante la realización del proyecto, se puede

destacar la curva de aprendizaje, es decir, la cantidad de conceptos y tecnologías nuevas que hemos tenido que aprender antes de desarrollar el proyecto, como son: sistemas multiagente, plataforma JADE y sus protocolos, API de Google Maps, Subversion, librería SWT, etc. En concreto, se indican a continuación los detalles más complejos llevados a cabo en el proyecto:

- Modelado de los actores del escenario de emergencias médicas: decidir los comportamientos y características de cada actor que participa en el entorno simulado no ha sido una tarea sencilla. Cada uno de ellos debe tener una lista definida de acciones y mensajes que puede enviar al entorno, así como unos atributos que les definan de la forma más realista posible y un conocimiento de lo que pueden percibir y lo que no del entorno que les rodea. Todos estos conceptos fueron analizados y consensuados en el proceso de captura de requisitos que debía cumplir el sistema, de forma que se consiguiese un escenario de emergencias médicas realista y completo.

- Protocolo de comunicación: entre las posibles alternativas planteadas para implementar el protocolo de comunicación entre los agentes, se decidió usar un mecanismo en el cual cada agente interacciona con el resto a través de un entorno, implementado también como agente. De esta forma, cualquier acción o comunicación que tenga lugar en el sistema pasará a través de este agente, el cual forma parte del subsistema *Simulator* para poder tener comunicación entre el entorno y el simulador.

- Tecnología de visualizado: elegir la utilización de la librería SWT para implementar las interfaces gráficas no ha sido al azar. Antes de su utilización, se estuvieron estudiando otras posibles alternativas, como la biblioteca gráfica Swing de Java o el framework GWT (*Google Web Toolkit)* de Google. La elección de SWT se debió a que contiene un componente gráfico capaz de proporcionarnos las funciones de un navegador Web que requería la aplicación, el *Browser SWT*. A partir de entonces, se siguió un proceso de aprendizaje de la librería y de diseño de las

interfaces gráficas hasta tener implementado el sistema gráfico actual.

- La utilización de Google Maps para el cálculo de rutas: con la API de Google Maps se pueden desarrollar multitud de funcionalidades en los mapas que proporciona, pero saber cómo conseguir esas funcionalidades es una labor bastante compleja debido a la escasa documentación existente en su página Web oficial. Conseguir calcular rutas para los movimientos solicitados por los agentes resultó bastante sencillo ya que Google Maps proporciona mecanismos para ello, pero el mostrar los movimientos de esas rutas de forma visual requería conocimientos más avanzados de esta API y de funciones trigonométricas. Encontrar las funciones y ecuaciones apropiadas para saber colocar los iconos representativos de agentes en la posición del mapa correspondiente, así como para dibujar sobre el mapa polígonos y círculos, implicó un gran tiempo de cálculo y búsqueda de información.

- Sistema operativo: algunas de las tecnologías y herramientas usadas en nuestro proyecto son dependientes del sistema operativo que las use como, por ejemplo, la librería SWT. Por ello, la aplicación desarrollada sólo es compatible con el sistema operativo Windows.

4.1. Líneas futuras

Este proyecto realiza toda la funcionalidad especificada, ya que cumple con los objetivos impuestos al principio, recogidos en los requisitos funcionales del sistema. Aún así, el sistema desarrollado podría tener futuras extensiones para su mejora. Estas extensiones podrían estar enfocadas hacia los siguientes objetivos:

- Crear nuevos modelos de aplicación para la simulación de diferentes mecanismos de gestión de emergencias médicas. Estos serán representaciones de los modelos existentes en diferentes ciudades, ya que la manera en que se gestionan las emergencias suele ser diferente dependiendo de la ciudad. Existen modelos en los que

no existe un centro coordinador y son el resto de actores lo que deben lograr la coordinación, también hay modelos en los que se utilizan incentivos para realizar las asistencias médicas, etc. Además de la simulación de modelos existentes, crearemos modelos que no estén implantados en la actualidad para simularlos, de forma que se podrá comprobar el comportamiento que proporcionarían en una determinada ciudad.

- Algunas de las funcionalidades actuales del proyecto están implementadas en la propia aplicación. Un ejemplo de esta situación es cuando una ambulancia debe determinar a qué hospital trasladar un paciente. Esta tarea podría realizarse mediante servicios Web que proporcionen información sobre la eficiencia de los diversos hospitales. Por ello, se crearán diferentes servicios Web que aporten la funcionalidad necesaria para cada decisión de los agentes.

- Crear nuevos visualizadores para mostrar de forma gráfica el comportamiento de la simulación. Estos nuevos visualizadores ampliarán la información relativa a la simulación de manera que se podrán evaluar a un nivel de detalle más bajo las interacciones de los distintos agentes. Además, se podrán crear visualizadores utilizando tecnologías distintas a las utilizadas en este proyecto. Alguna de estas implementaciones consistirá en una aplicación web en la que se visualizará la simulación.

- Una de las lineas más importantes a seguir es la creación de un módulo de evaluación para poder evaluar el comportamiento de cada modelo de emergencias. Para ello, este nuevo subsistema almacenará los datos relativos a todas las simulaciones de manera que se evalúe la eficiencia de cada modelo. Esto permitirá determinar qué ventajas e inconvenientes proporciona cada uno de ellos, pudiendo así mejorar los distintos modelos. Entre los datos que será capaz de analizar, se encuentran los relativos al tiempo empleado en realizar una asistencia medica, la eficiencia con la que se ha realizado, el porcentaje de pacientes curados, el coste total de los recursos empleados, etc.

- Utilizar datos reales de distintas ciudades del mundo para las simulaciones: en nuestro proyecto, la simulación de un entorno de emergencias se desarrolla mediante distribuciones estadísticas predefinidas, es decir, algunos de los datos relevantes de la simulación tales como el número de llamadas por unidad de tiempo en el centro coordinador nos es desconocido, por lo que se genera de acuerdo a estas distribuciones. Para aportar mayor realismo al sistema, todos los datos relativos a las emergencias médicas deberían ser conocidos. Entre estos datos se encuentran, por ejemplo, el número total de ambulancias y su diferentes zonas de acción. Por ello, una de las lineas futuras más importantes a seguir es la obtención de toda la información relativa a emergencias médicas de los distintos modelos de aplicación que se implementen. Esta línea actualmente está en desarrollo, debido a que existen reuniones programadas con el SUMMA de la Comunidad de Madrid para aportarnos todo tipo de datos reales.

Anexo A Manual de usuario

A.1. Arrancando el demostrador mHealth

Para comenzar a usar el demostrador se puede lanzar la aplicación de dos maneras distintas:

- A partir de una consola, escribimos la siguiente sentencia situándonos en el directorio *mHealthDemostrator*: `java -Xmx1024M -jar mHealthDemostrator.jar`
- Hacer doble click en el ejecutable ***mHealthDemostrator.bat***

Los ficheros utilizados para ambas ejecuciones se proporcionan en el CD-Rom de esta memoria.

Cuando inicializamos el demostrador *mHealth*, nos encontramos con la ventana principal que podemos ver en la figura A.1. Se puede acceder a toda la funcionalidad del demostrador a través del menú de esta ventana, donde se muestran las tres opciones del sistema:

- Crear la configuración de una simulación.
- Ejecutar una simulación desde un fichero de configuración.
- Visualizar una simulación previamente creada.

Tras la elección de la opción deseada, pulsar *Accept* para ejecutar esa funcionalidad. De lo contrario, si se pulsa el botón *Cancel*, se cerrará la aplicación.

A.2. Crear la configuración de una simulación

Para crear una configuración nueva, se utiliza el módulo *Setup* del demostrador *mHealth*. Este módulo es el encargado de mostrar al usuario una serie de ventanas que le irán solicitando los parámetros de la configuración necesarios para poder ejecutar después una simulación. Los pasos a seguir para la creación de la configuración son:

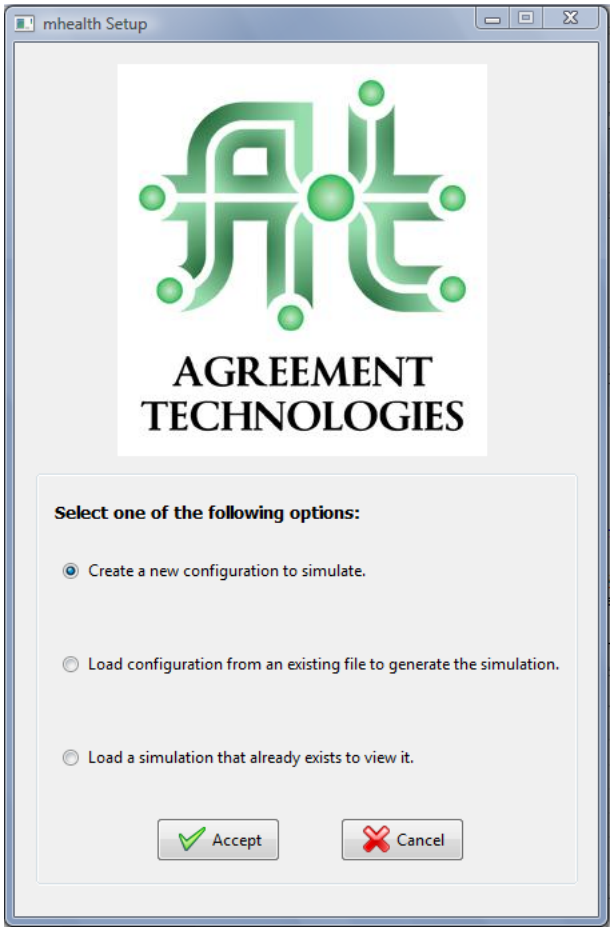


Figura A.1: Ventana principal de mHealth

1. Seleccionar la opción "*Create a new configuration to simulate*" de la ventana principal del demostrador A.1 y pulsar *Accept*.

2. Introducir información global de la simulación: la ventana A.2 es la primera que se muestra en el *mHealth Setup*. En ella, se deben indicar los siguientes parámetros:

   - *Simulation identifier*: es el nombre de la configuración que se va a crear. Ese nombre se usará para la generación del fichero de configuración y del fichero histórico que se genere en la simulación.
   - *Configuration file path*: es la ruta donde se quiere guardar el fichero de configuración. Por defecto, será la misma ubicación donde está el ejecutable

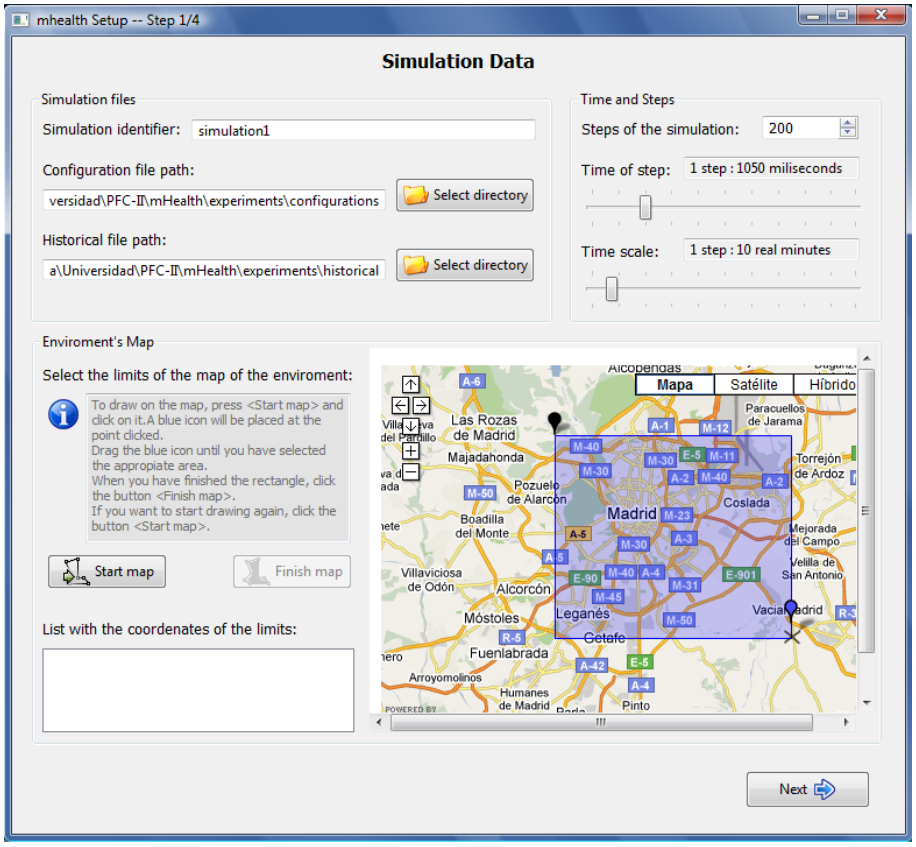


Figura A.2: Ventana inicial del mHealth Setup

de la aplicación. Si esta dirección se quiere cambiar, se debe pulsar el botón *Select directory* y especificar el destino del fichero de configuración, o escribir manualmente la ruta deseada.

- *Historical file path*: es la ruta donde se quiere guardar el fichero histórico que se genere en la simulación. Por defecto, será la ubicación donde está el ejecutable de la aplicación. Si esta dirección se quiere cambiar, se debe pulsar el botón *Select directory* y especificar la ruta destino del fichero histórico, o escribir manualmente la ruta deseada.

- *Steps of the simulation*: es el número total de pasos que tendrá la simulación. Cada uno de esos pasos, será una iteración del protocolo acción-percepción definido para los agentes software de la aplicación.

- *Time of step*: es la duración de cada uno de los pasos que forman la simulación, en milisegundos.
- *Time scale*: es la escala de tiempo que se aplicará en los movimientos de los agentes en el entorno. Cada paso de simulación, se corresponderá con tantos minutos reales como se indiquen en esta escala.
- *Environment's map*: el entorno de la simulación tendrá un territorio delimitado en el que podrán actuar el resto de actores del escenario de emergencias. Este territorio se especifica en el mapa de este área. Para comenzar su limitación, se tiene que pulsar el botón *Start*. Una vez pulsado, al pasar el cursor del ratón por encima del mapa se debe notar que éste se transforma a un aspa. En ese momento, podemos hacer click sobre el mapa y un marcador azul se colocará en la posición seleccionada. Para dibujar el rectángulo delimitador del mapa, arrastrar ese marcador azul hasta tener el área deseada. Al soltar el marcador, el mapa quedará limitado y se pondrá visible el botón *Finish*. Si queremos guardar los cambios hechos en el mapa, se debe pulsar ese botón y veremos cómo se añade una lista de coordenadas geográficas debajo de los botones tal y como muestra la ventana A.3. Si por el contrario, queremos deshacer el rectángulo dibujado, se debe pulsar de nuevo el botón *Start* y comenzar el proceso explicado anteriormente.

Una vez completados todos los parámetros anteriores, pulsar el botón *Next*. Si algún campo se quedó sin completar, el sistema nos mostrara el error "*You must fill all the information to go to the next step*". En ese caso, se debe proceder a rellenar el campo faltante y continuar con la creación de la configuración pulsando *Next*.

3. Indicar los parámetros de los centros de emergencias, pacientes y distribuciones estadísticas: la siguiente pantalla del *Setup* es la mostrada en A.4. En ella, se deben especificar los siguientes datos:

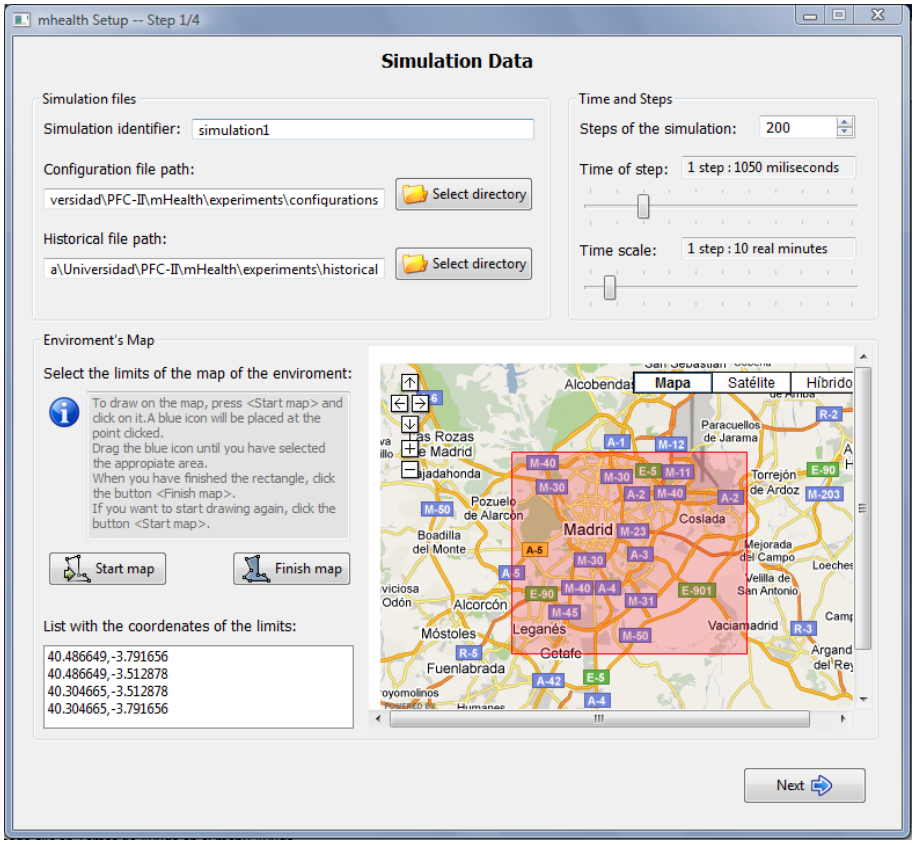


Figura A.3: Ventana inicial del mHealth Setup completada

a) Centros de emergencia: los parámetros que definen a cada centro de emergencias son:

- *Alias*: es el nombre público que recibirá el centro coordinador especificado.
- *Number of call operators*: es el número de operadores telefónicos que tendrá el centro especificado. Como mínimo, debe tener un telefonista.

Cuando se indique todos los parámetros, se tiene que pulsar el botón *Add* para guardar el centro de emergencia configurado. Si el alias del centro no se ha indicado, se mostrará el mensaje de error "*The identifier of the emergency center is empty*" y no se guardarán los datos. Si por el contrario, todos los campos están especificados pero ya existe un centro con el mismo

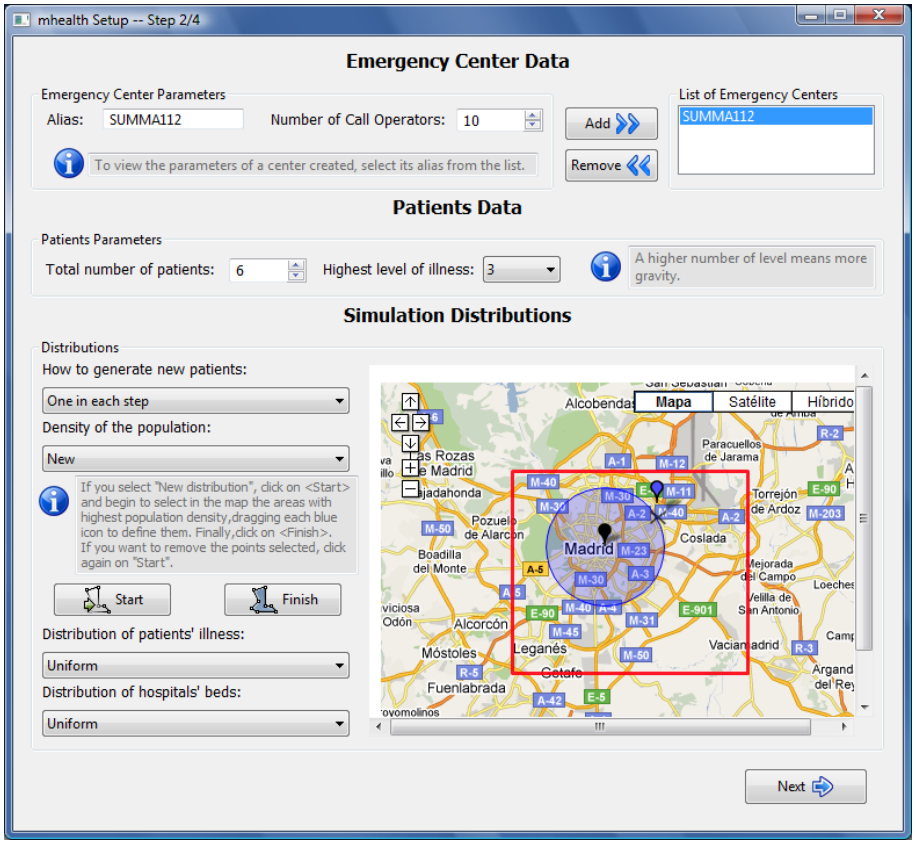


Figura A.4: Formulario para la creación de centros de emergencia y pacientes

alias, se informará al usuario con el error "*The identifier of the emergency center already exists in the list*" y no se almacenará el centro definido. Una vez estén los datos correctos, tras pulsar *Add* se guardará el centro creado y se mostrará en la lista de la derecha. Para visualizar el contenido de un centro de la lista, bastará con seleccionar su alias. Si se desea eliminar el centro seleccionado, se debe pulsar el botón *Remove* y éste se borrará de la lista de centros coordinadores.

b) Pacientes: los datos relativos a los pacientes son el número total de enfermos que se generarán durante la simulación (*Total number of patients*) y el mayor nivel de enfermedad que puede tener un paciente (*Highest level of illness*). Este último parámetro, como máximo podrá valer 5.

c) Distribuciones estadísticas: el usuario puede elegir entre diversas distribuciones para cada concepto de la simulación que se va a modelar aleatoriamente. Dichos conceptos son los siguientes:

- *How to generate new patients*: las distribuciones implementadas para la generación de pacientes son *Uniform* (los pacientes se crearán a lo largo de la simulación de manera uniforme), *One in each step* (durante el comienzo de la simulación, se crearán todos los pacientes de uno en uno en cada *timestep*) y *All in the first step* (el número total de pacientes definido se generarán en el primer paso de la simulación).
- *Density of the population*: hay dos posibles distribuciones, *Uniform* que considera todos los puntos del mapa del entorno equiprobables para ser seleccionados como ubicación de un paciente y *New*, que permite al usuario definir puntos de alta densidad de población sobre el mapa. Para definir esos puntos, se debe elegir esa distribución y el botón situado debajo, llamado *Start*, se habilitará. Al pulsarlo, el cursor del ratón se pondrá con forma de aspa al situarse encima del mapa. Se debe pinchar en el lugar del mapa que se desee poner como punto de alta densidad demográfica, y tras la colocación de un marcador azul, se debe arrastrar hasta completar el círculo que limitará el área de alta población. Se pueden añadir tantos puntos como se deseen. Cuando se desee finalizar, se debe pulsar el botón *Finish*. Los puntos de densidad se cambiarán a color rojo para indicar que se han almacenado en la configuración, tal y como se aprecia en la figura A.5. Si se quieren borrar y definir una nueva población, se deberá volver a elegir la distribución *New* y puslar *Start*.
- *Distribution of patients' illness*: se han implementado las distribuciones *Uniform* y *Exponential* para la distribución de los niveles de enfermedad en los pacientes. En la primera, los niveles de enfermedad son equiprobables en toda la población. En la segunda, se define una prob-

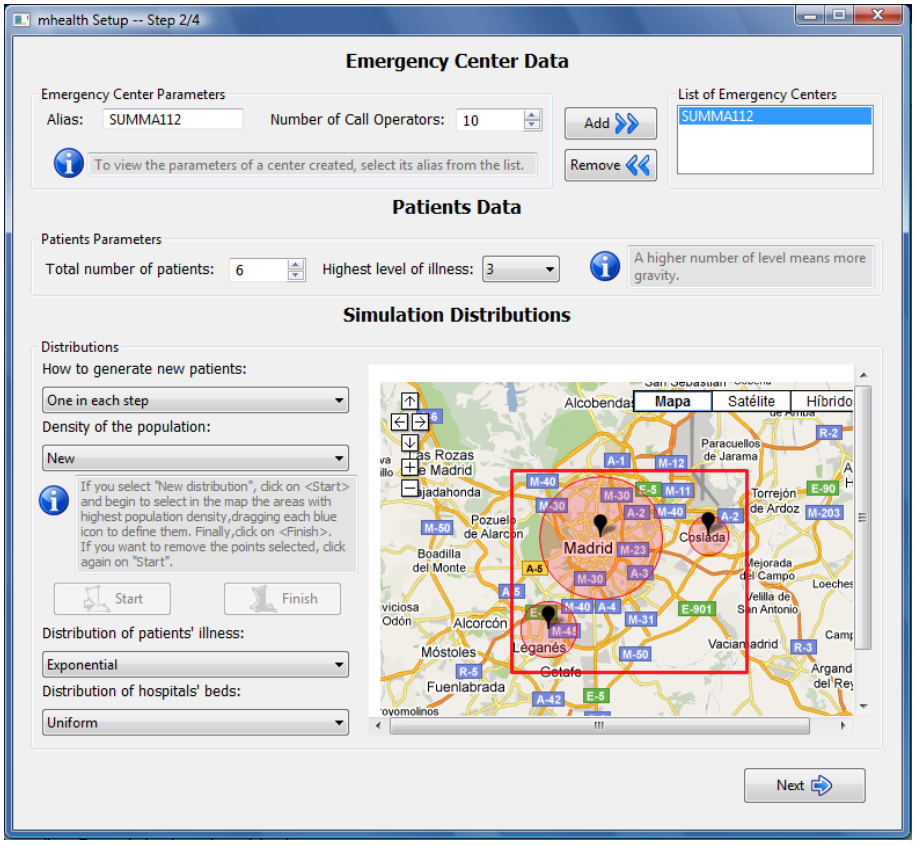


Figura A.5: Formulario para la creación de centros de emergencia y pacientes completado

abilidad para cada enfermedad, de forma que un paciente tenga mayor probabilidad de tener un nivel de enfermedad leve que uno grave de forma exponencial.

- *Distribution of hospitals' beds*: la ocupación de las camas de los hospitales puede ditribuirse con *Uniform* (la tasa de ocupación será uniforme a lo largo de la simulación, pudiendo tener cualquier porcentaje de ocupación en cualquier momento) y *Normal* (la tasa de ocupación tendrá su punto álgido en torno a la mitad de la simulación, aumentando su porcentaje hasta ese punto y disminuyendo después).

Completados todos los datos solicitados, pulsar el botón *Next*. Si aparece un error con el mensaje "*You must create at least one emergency Center*", es porque no se configuró ningún centro de emergencia y para continuar, todos los datos deben estar completos. El resto de parámetros, al tener valores por defecto, no generarán errores.

4. Describir las ambulancias del entorno: el siguiente formulario que se solicita es el mostrado en la figura A.6. En él se muestran los atributos de las ambulancias que se deben rellenar para configurar las que aparecerán en la simulación. Existen tres formas posibles para crear ambulancias, y las tres se pueden combinar:

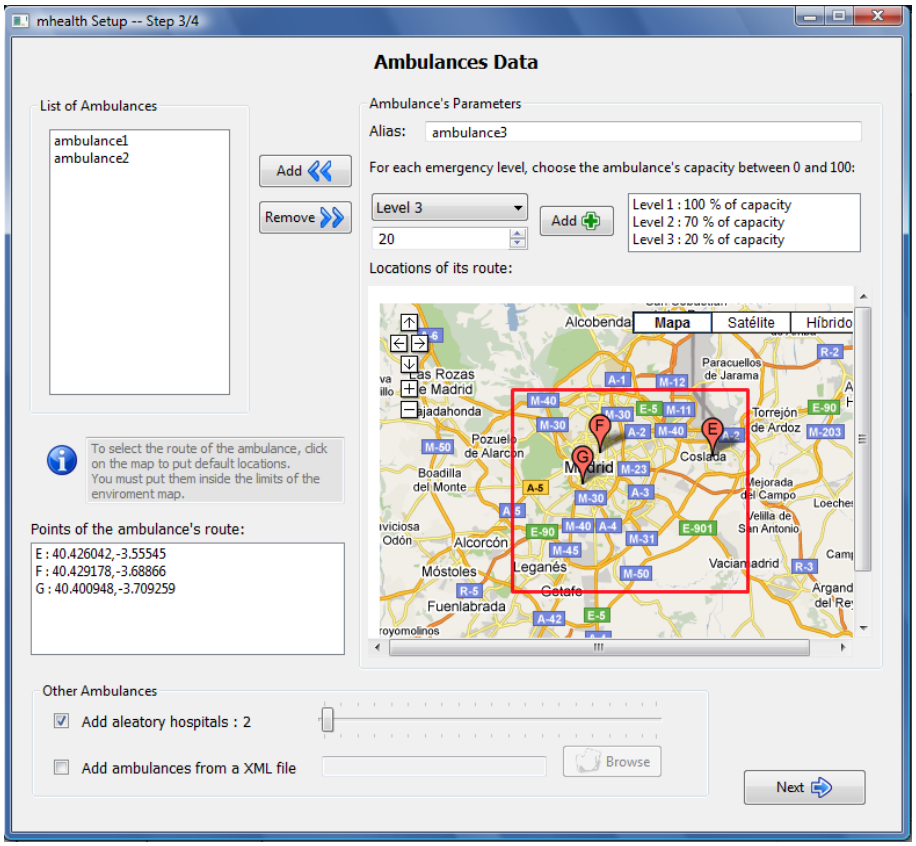


Figura A.6: Formulario para la creación de ambulancias

a) Ambulancias definidas por el usuario: se pueden crear ambulancias especificando los siguientes parámetros:

- *Alias*: es el nombre público que tendrá la ambulancia durante la simulación.
- *Ambulance capacity*: por cada nivel de enfermedad definido, se debe elegir el porcentaje de adecuación que tendrá la ambulancia para asistir a un paciente de ese nivel. Cada vez que se defina una capacidad, se debe añadir a la lista de la derecha con el botón *Add* verde. Una vez añadido, no se podrá borrar la capacidad guardada.
- *Locations of its route*: para indicar la ruta de posibles movimientos de una ambulancia cuando está libre, se deben elegir puntos geográficos en el mapa mostrado pinchando sobre él. Cada punto seleccionado, se mostrará en la lista *Points of the ambulance's route*. Una vez definido un lugar, éste no se puede eliminar.

Una vez completados esos datos, se puede guardar la ambulancia creada pulsando el botón *Add* azul. Su alias se añadirá a la lista de la izquierda de la ventana. Si algún parámetro se ha omitido, saltará el error "*The ambulance hasn't got all its parameters completed*".

b) Ambulancias aleatorias: se pueden definir ambulancias aleatorias seleccionando la opción *Add aleatory ambulances* del área inferior de la ventana. Una vez seleccionada la opción, se puede desplazar la barra de la derecha para elegir el número de ambulancias que se crearán aleatoriamente.

c) Ambulancias cargadas desde fichero: el usuario puede elegir un fichero XML con datos de ambulancias si selecciona la última opción del área inferior de la ventana. Se activará el botón *Browse*, el cual permite buscar la ruta del fichero de ambulancias que se desea cargar. Si el fichero elegido no contiene información de ambulancias en formato válido, se mostrará el error "*The file selected hasn't got a valid format*" y su contenido no se cargará en el sistema. Si de lo contrario, el fichero seleccionado es válido, se añadirán los alias de las ambulancias a la lista general de ambulancias configuradas.

Para visualizar el contenido de una ambulancia en concreto, simplemente se debe seleccionar el alias de la lista y se mostrarán sus datos en los campos de la derecha. Si se desea borrar la ambulancia seleccionada, se debe pulsar el botón *Remove* y su alias desaparecerá de la lista.

Finalmente, se debe pulsar el botón *Next* para proseguir con la creación de la configuración. Si ninguna ambulancia fue configurada, ni se seleccionó la creación de ambulancias aleatorias ni cargadas desde fichero, se mostrará el mensaje de error "*You must create at least one ambulance for the simulation*" y no se permitirá al usuario pasar a la siguiente ventana hasta que no defina, por lo menos, una ambulancia.

5. Especificar los parámetros de los hospitales del escenario médico: el último formulario que se solicita es el mostrado en la figura A.7. En él, se muestran los atributos de los hospitales que se deben rellenar para configurar los que aparecerán en la simulación. Existen tres formas posibles para crear hospitales, y las tres se pueden combinar:

   a) Hospitales definidos por el usuario: se pueden crear hospitales especificando los siguientes parámetros:

   - *Alias*: es el nombre público que tendrá el hospital durante la simulación.
   - *Maximum level of urgency*: nivel máximo de enfermedad de pacientes que es capaz el hospital de atender en su área de urgencias médicas.
   - *Number of beds*: número total de camas del hospital.
   - *Number of members in the emergency area*: cantidad de personal sanitario trabajador en el área de urgencias del hospital que se está configurando.
   - *Location*: ubicación del hospital dentro del mapa del entorno. Para indicar su localización, basta con pinchar en el punto exacto del mapa donde se va a ubicar. Si se quiere concretar mejor, se puede desplazar el icono azul creado hasta la dirección deseada.

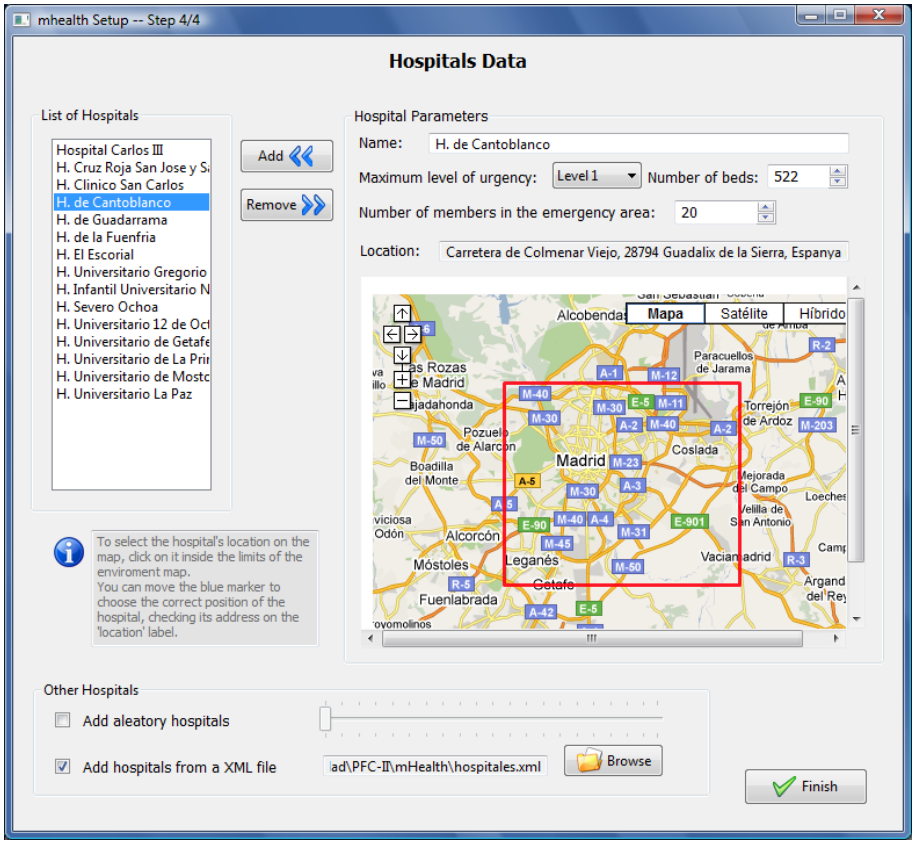


Figura A.7: Formulario para la creación de hospitales

Una vez completados esos datos, se puede guardar el hospital creado pulsando el botón *Add*. Su alias se añadirá a la lista general de hospitales. Si algún parámetro se ha omitido, saltará el error "*Missing values for the hospital*".

b) Hospitales aleatorios: se pueden definir hospitales aleatorios seleccionando la opción *Add aleatory hospitals* del área inferior de la ventana. Una vez seleccionada la opción, se puede desplazar la barra de la derecha para elegir el número de hospitales que se crearán aleatoriamente en el entorno.

c) Hospitales cargados desde fichero: el usuario puede elegir un fichero XML con datos de hospitales si selecciona la última opción del área inferior de la ventana. Se activará el botón *Browse*, el cual permite buscar la ruta del

fichero de hospitales que se desea cargar. Si el fichero elegido no contiene información de hospitales en formato válido, se mostrará el error "*The file selected hasn't got a valid format*" y su contenido no se cargará en el sistema. Si de lo contrario, el fichero seleccionado es válido, se añadirán los alias de los hospitales a la lista general de hospitales configurados.

Para visualizar el contenido de un hospital en concreto, simplemente se debe seleccionar su alias de la lista y se mostrarán sus datos en los campos de la derecha. Si se desea borrar el hospital seleccionado, se debe pulsar el botón *Remove* y su alias desaparecerá de la lista.

Finalmente, se debe pulsar el botón *Finish* para concluir la configuración de la simulación. Si ningún hospital fue configurado en esta pantalla, ni se seleccionó la creación de hospitales aleatorios ni cargados desde fichero, se mostrará el mensaje de error "*You must create at least one hospital for the simulation*" ya hasta que no se defina por lo menos una hospital, no se permitirá al usuario finalizar el modulo *Setup*.

Terminada la configuración de una simulación tras tener todos los parámetros solicitados completados, el sistema muestra la ventana A.8. En ella, se muestra al usuario la ubicación de los ficheros configuración e histórico de la simulación que se va a ejecutar. Además, se le pregunta al usuario si quiere visualizar lo que va sucediendo en la simulación a la vez que se realiza (opción *Show the simulation with a graphical interface*), funcionalidad implementada en el subsistema *Visualizador*. Si desea habilitar esa funcionalidad, deberá marcar el recuadro de su izquierda tal y como se ve en la imagen A.8.

Otra opción implementada es la visualización de la interfaz gráfica de la plataforma JADE, para poder ver como interactúan los agentes, qué mensajes se envían, en qué máquinas se generan, etc. Si se elige esta interfaz, no se podrá elegir la del subsistema *Visualizador* y viceversa, ya que sólo puede estar activa una única interfaz durante la simulación.

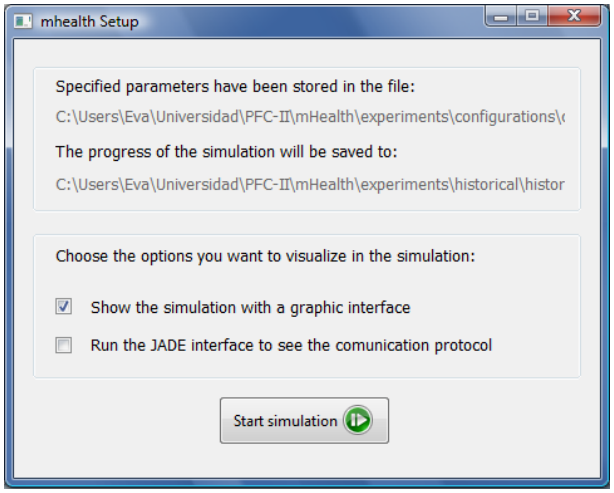


Figura A.8: Ventana final del módulo *Setup*

Para comenzar la ejecución de la simulación, pulse *Start Simulation*.

A.3. Ejecutar una simulación

A esta funcionalidad se llega desde dos opciones de la ventana principal: eligiendo crear una configuración, opción que finalizará con la ejecución de la simulación configurada, o eligiendo ejecutar una simulación en la ventana principal de *mHealth*. A continuación, se explican los pasos que se deben seguir par ejecutar una simulación con la segunda opción:

1. Seleccionar la opción "*Load a configuration from an existing file to generate the simulation*" de la ventana principal del demostrador y pulsar *Accept*.

2. Seleccionar el fichero de configuración que se desea cargar, buscando la ruta donde se encuentra. Si el fichero seleccionado no contiene el formato correcto de una configuración, se mostrará un error con el siguiente texto: "*The configuration selected can not be loaded. Select another one*". Tras aceptar el error, se volverá a la pantalla de inicio del demostrador para poder elegir de nuevo la funcionalidad que se desee realizar.

3. Si se ha seleccionado un fichero de configuración correcto, nos aparecerá la ventana final del *mHealth Setup* A.8. En ella, se muestra al usuario la ubicación del fichero configuración seleccionado y la ruta donde se almacenará el fichero histórico de la simulación que se va a ejecutar. Además, se le pregunta al usuario si quiere visualizar lo que va sucediendo en la simulación a la vez que se realiza (opción *Show the simulation with a graphical interface*), funcionalidad implementada en el subsistema *Visualizador*. Si desea habilitar esa funcionalidad, deberá marcar el recuadro de su izquierda tal y como se ve en la imagen A.8. Para comenzar la simulación, puslar el botón *Start Simulation*.

4. En este momento, dependiendo de la opción de visualización seleccionada por el usuario en el paso anterior, se mostrará una de las siguientes ventanas:

   a) Ventana del subsistema *Visualizador*: la pantalla A.9 aparece cuando se selecciona la opción de visualizar la simulación de forma gráfica. Esta ventana se compone de cuatro áreas de información:

   - Área superior: Barra de progreso.

     Esta zona se compone de una barra de progreso que nos muestra el porcentaje de simulación realizado. Además, se indica en forma de texto el número de pasos totales que tiene la simulación y el número de pasos que se han realizado en cada momento. Cuando la simulación finalice, esta barra de progreso se completará al 100 % y el resto de la áreas de la ventana se quedarán con la última información recibida desde el simulador.

   - Área de la izquierda: navegador Web.

     En el navegador Web de la ventana se muestra un mapa de Google Maps con los movimientos y posiciones de los diversos agentes que actúan en el escenario de emergencias médicas. Para desplazarnos por el mapa, basta con situar el cursor del ratón sobre él, pinchar y arrastrar para mover el área del mapa visualizado. Además, este mapa tiene una

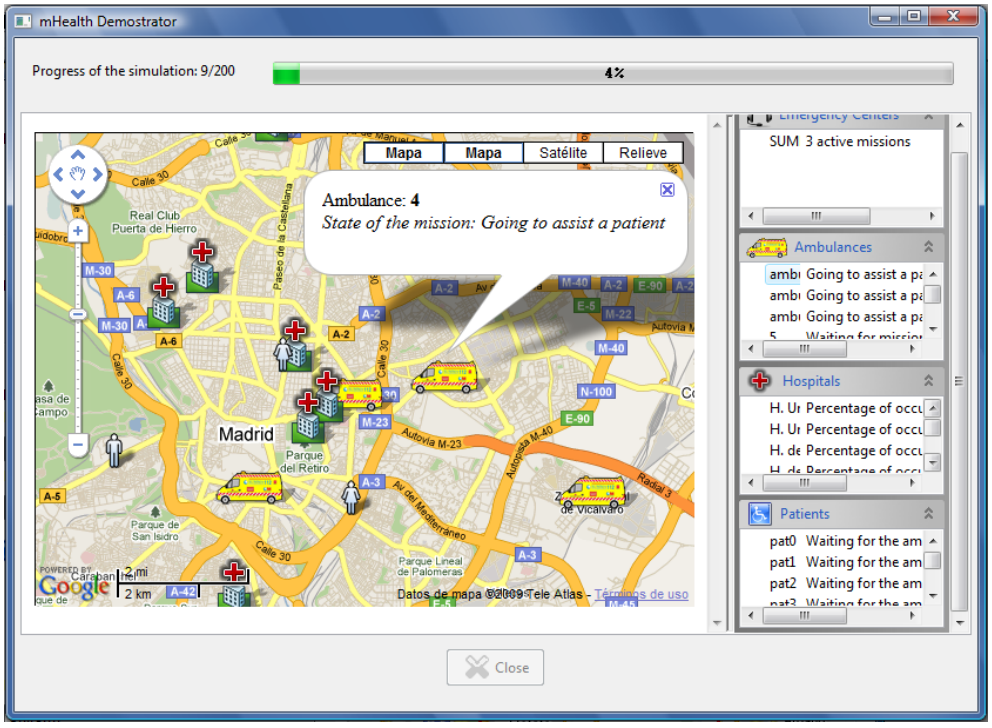


Figura A.9: Ventana del *Visualizador* con los datos globales de la simulación

barra de zoom en su esquina superior izquierda que permite alejarse o alejarse de la zona visualizada.

Los iconos que representan a los agentes de la aplicación son:

- Hospitales  : indican la ubicación de cada hospital en el mapa del entorno.
- Ambulancias sin pacientes  : muestran la posición de cada ambulancia del sistema en cada momento y sus movimientos. Si la ambulancia tiene esta apariencia, es porque no contiene ningun paciente en su interior.
- Ambulancias con pacientes  : muestran la posición de

cada ambulancia del sistema en cada momento y sus movimientos. Si la ambulancia tiene esta apariencia, es porque tiene a un paciente en su interior al que está trasladando hacia un hospital.

- Pacientes enfermos  : ambos iconos representan pacientes enfermos en el entorno, indicando sus posiciones sobre el mapa.
- Pacientes que reciben asistencia *in situ*  : ambos iconos forman parte de las animaciones que aparecen cuando un paciente enfermo está recibiendo asistencia *in situ*. El cuerpo del icono se va rellenando de color verde según avance la asistencia médica.
- Personas sanas  : ambos iconos aparecen cuando los pacientes, tras recibir asistencia, se curan.
- Personas muertas  : ambos iconos aparecen cuando los pacientes fallecen.

Cada vez que pasemos el cursor del ratón por encima del marcador de un agente, se mostrará un globo con información del estado de dicho agente en ese momento, tal y como se ve en la imagen A.9.

- Área de la derecha: panel informativo con los agentes que participan en el entorno.

  Este panel se compone de cuatro áreas, una por cada tipo de actor que participa en el entorno, que pueden expandirse o contraerse para visualizar su lista interna de agentes o no, respectivamente:

  - Centros de emergencias  : en este panel se muestra un listado con los diferentes centros coordinadores que existen en el

escenario de la simulación. De cada uno de ellos, se muestra su alias y el número total de misiones activas que está gestionando.

- Ambulancias  : contiene un listado de las ambulancias que participan en el entorno, indicando sus alias y el estado de la misión por el que pasan en cada *timestep* resumido en una línea de texto.
- Hospitales  : este panel tiene una lista con todos los hospitales del entorno, indicando sus alias y la tasa de ocupación de sus camas en cada paso de la simulación.
- Pacientes  : contiene un listado de los pacientes (enfermos, sanos o muertos) que participan o han participado en el entorno, indicando sus identificadores y el estado por el que pasan en cada *timestep*, resumido en una línea de texto.

La información de estos paneles se irán actualizando en cada paso de la simulación. Para ampliar la información asociada a cada agente del entorno, se puede seleccionar del panel anterior el identificador del agente que se quiere detallar. Una vez se pincha sobre su nombre, aparece una ventana modal con la información individual del agente seleccionado, pudiendo obtener ventanas como las mostradas en la figura A.10. En ellas, aparecen: el alias del agente, un icono representativo del tipo de actor que es, un área de texto superior con el estado interno del paciente en el *timestep* que se abrió la ventana, y un área de texto inferior con el histórico de acciones realizadas por el agente junto con el número de *timestep* en el que se ejecutó cada acción.

En cada ventana modal, se puede actualizar la información del agente

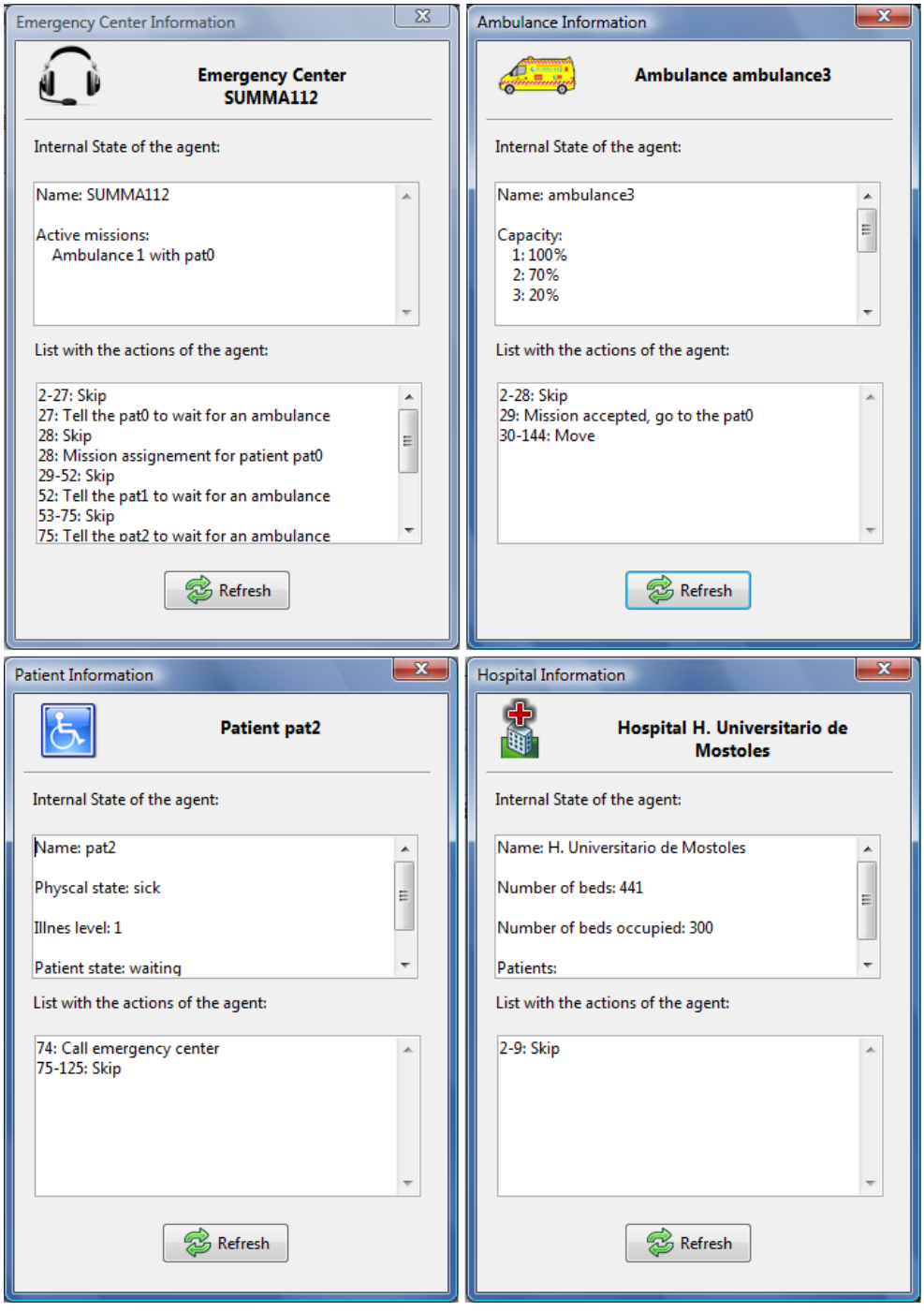


Figura A.10: Ventanas del *Visualizador* con los datos individuales de los actores

asociado pulsando el botón *Refresh*. Las áreas de texto modificarán su contenido y mostrarán el estado interno actual y el nuevo histórico de acciones del agente.

Al poder abrir varias ventanas de un mismo agente, se pueden ir viendo las diferencias entre los estados internos de dicho agente de cada *timestep* elegido, mientras que en una sola ventana se puede ver la evolución de las acciones del agente al ir añadiéndose cada acción nueva al histórico de sus acciones.

- Área inferior: botón de finalización.

  Mientras la simulación se va procesando, éste botón permanece inhabilitado para impedir que el usuario cierre el subsistema *Visualizador*. Una vez se ha concluido la simulación, se activa el botón y el usuario puede cerrar esta ventana para volver al menú inicial del demostrador *mHealth*.

b) Ventana con el progreso de la simulación: cuando el usuario no quiere que se muestre gráficamente la ejecución de la simulación, aparece la ventana A.11. En ella se muestra una barra de progreso que se va completando según la simulación va avanzando.

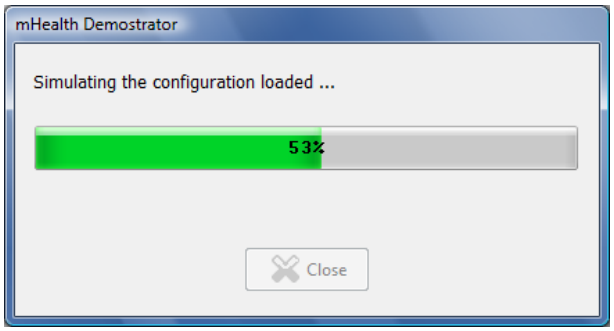


Figura A.11: Ventana de progreso a mitad de una simulación

Una vez terminada la simulación, la barra de progreso se completa al 100 % y se activa el botón *Close* para poder volver a la ventana de inicio del

mHealth, tal y como se ve en la figura A.12 .

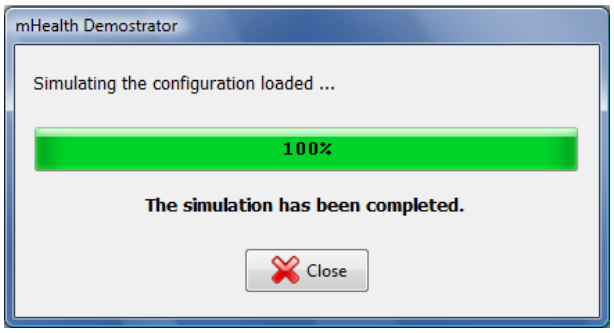


Figura A.12: Ventana de progreso con la simulación finalizada

5. Finalizada la ejecución de la simulación, la barra de progreso se completará y se habilitará el botón inferior *Close*, se haya usado la interfaz del *Visualizador* o la ventana de progreso del simulador. Pulsando ese botón, se cierra la ventana actual y se vuelve al menú inicial del demostrador *mHealth* (figura A.1).

A.4. Visualizar una simulación previamente creada

La última opción mostrada en la pantalla principal de *mHealth*, es la encargada de permitir al usuario ver una simulación que se ejecutó previamente y cuyo fichero histórico fue almacenado y completado, de forma gráfica, sin necesidad de volver a ejecutar la configuración asociada a esa simulación. Los pasos necesarios para visualizar la simulación son:

1. Seleccionar la opción "*Load a simulation that already exists to view it*" de la ventana principal del demostrador y pulsar *Accept*.

2. Seleccionar el fichero con el histórico de la simulación que se desea cargar, buscando la ruta donde se encuentra. Si el fichero seleccionado no contiene el formato correcto de un histórico, se mostrará un error con el siguiente texto: "*The historical selected can not be loaded. Select another one*". Tras aceptar el error, se

volverá a la pantalla de inicio del demostrador para poder elegir de nuevo la funcionalidad que se desee realizar.

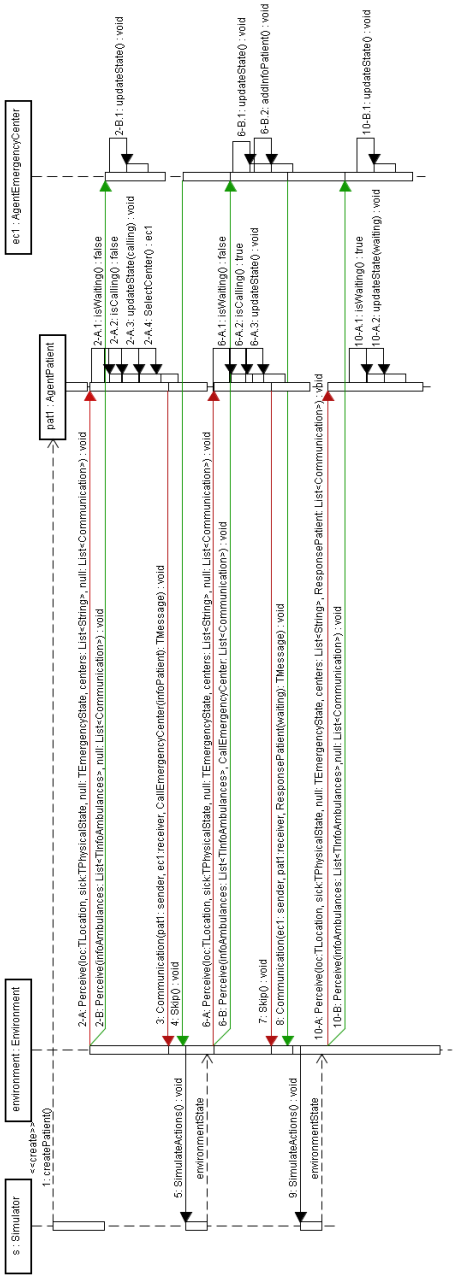
3. Si se ha cargado un fichero correcto, comenzará la visualización de su contenido. La ventana que mostrará la simulación de forma global es la capturada en la figura A.9. Su utilización es igual que la descrita en la sección "Ejecutar una simulación", punto $4a$, donde se explica cómo usar el panel de información de los agentes, el mapa del navegador y las ventanas con información individual de cada agente.

4. Finalizada la lectura del fichero de la simulación, la barra de progreso superior se completará y se habilitará el botón inferior *Close*. Pulsando ese botón, se cierra la ventana del *Visualizador* y se vuelve al menú inicial del demostrador *mHealth* (figura A.1) para poder seguir usando el programa o cerrarlo.

Anexo B Diagramas de secuencia

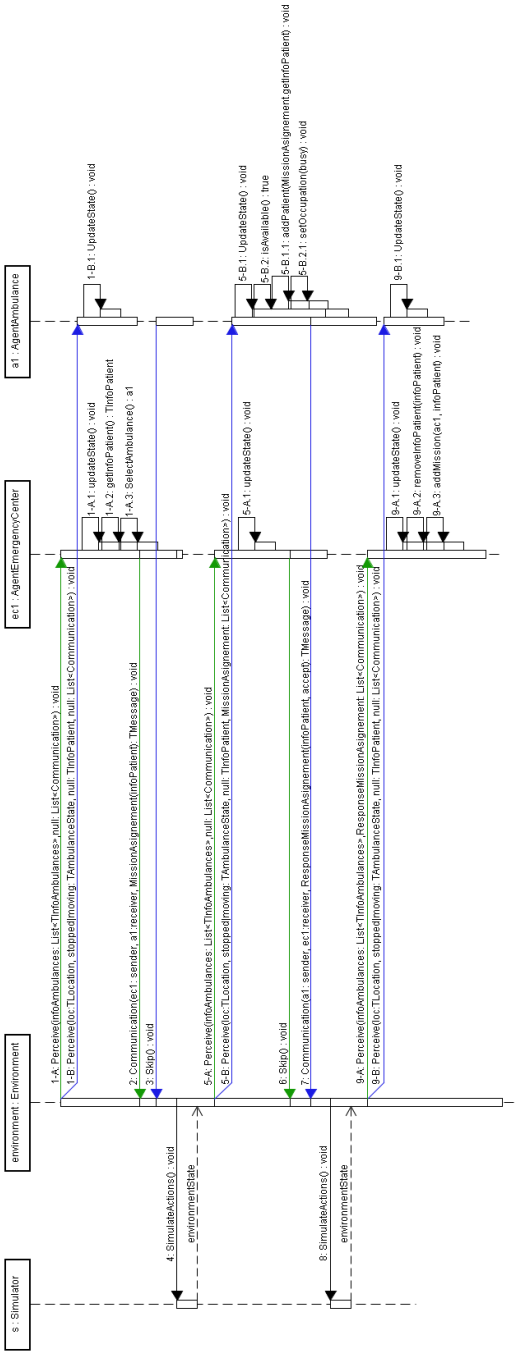
B.1. Aplicación

B.1.1. Comunicación de paciente con centro de emergencias

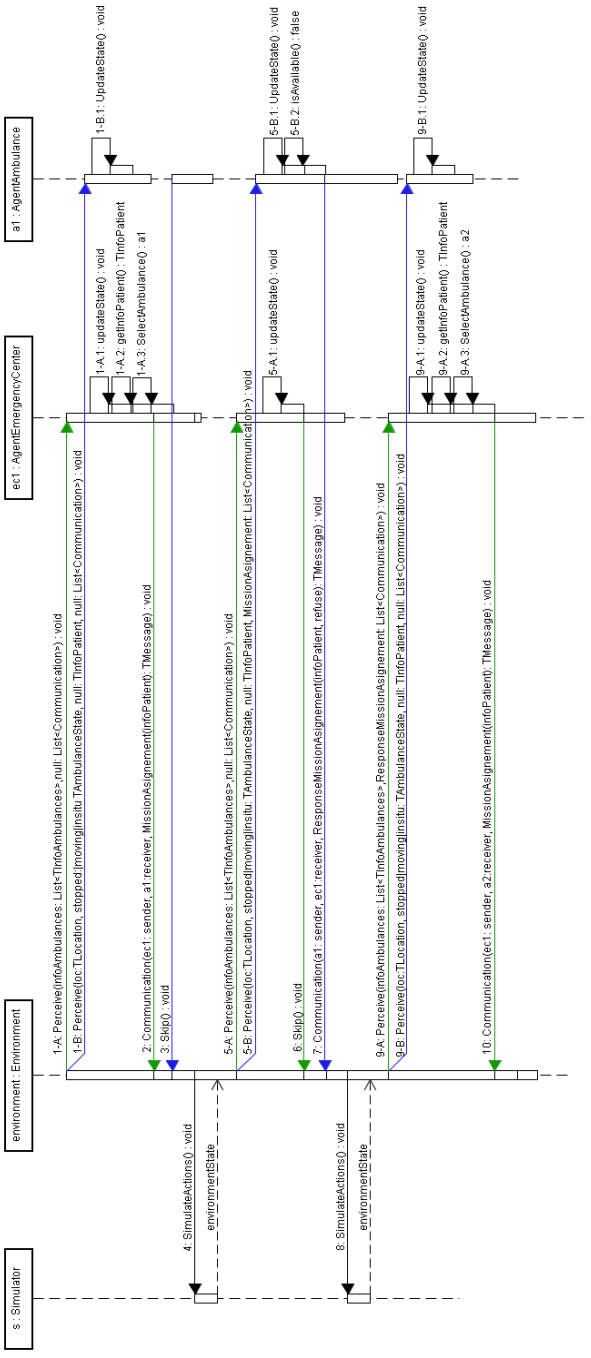


B.1.2. Comunicación de centro de emergencias con ambulancia

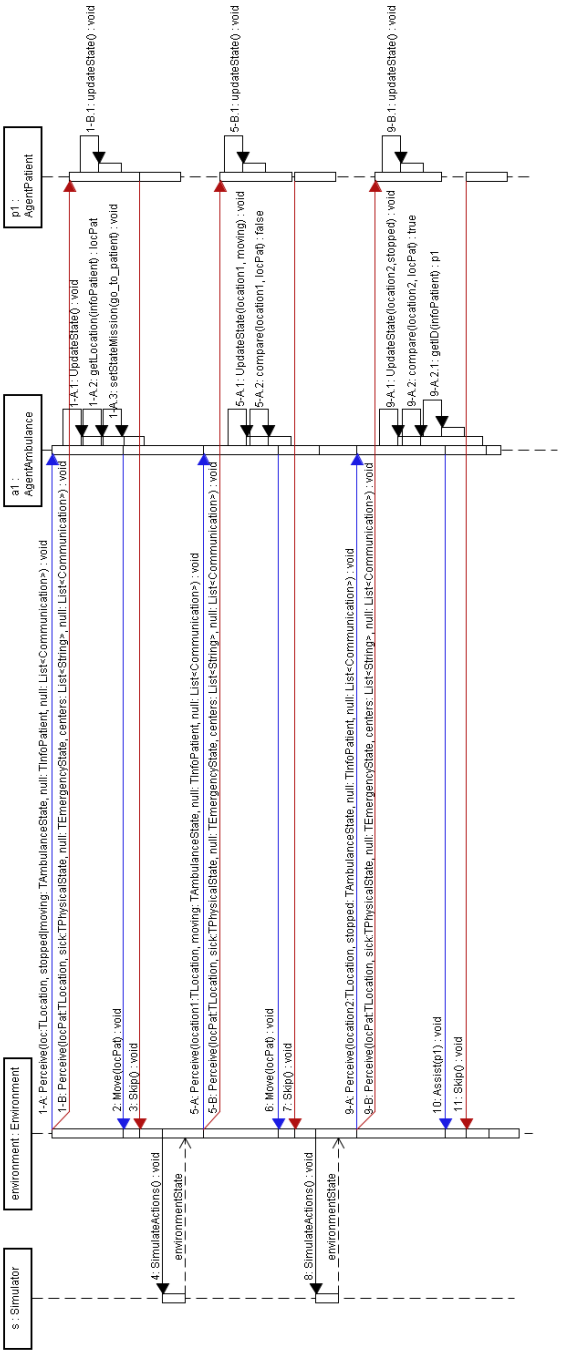
B.1.2.1. Acepta



B.1.2.2. Rechaza

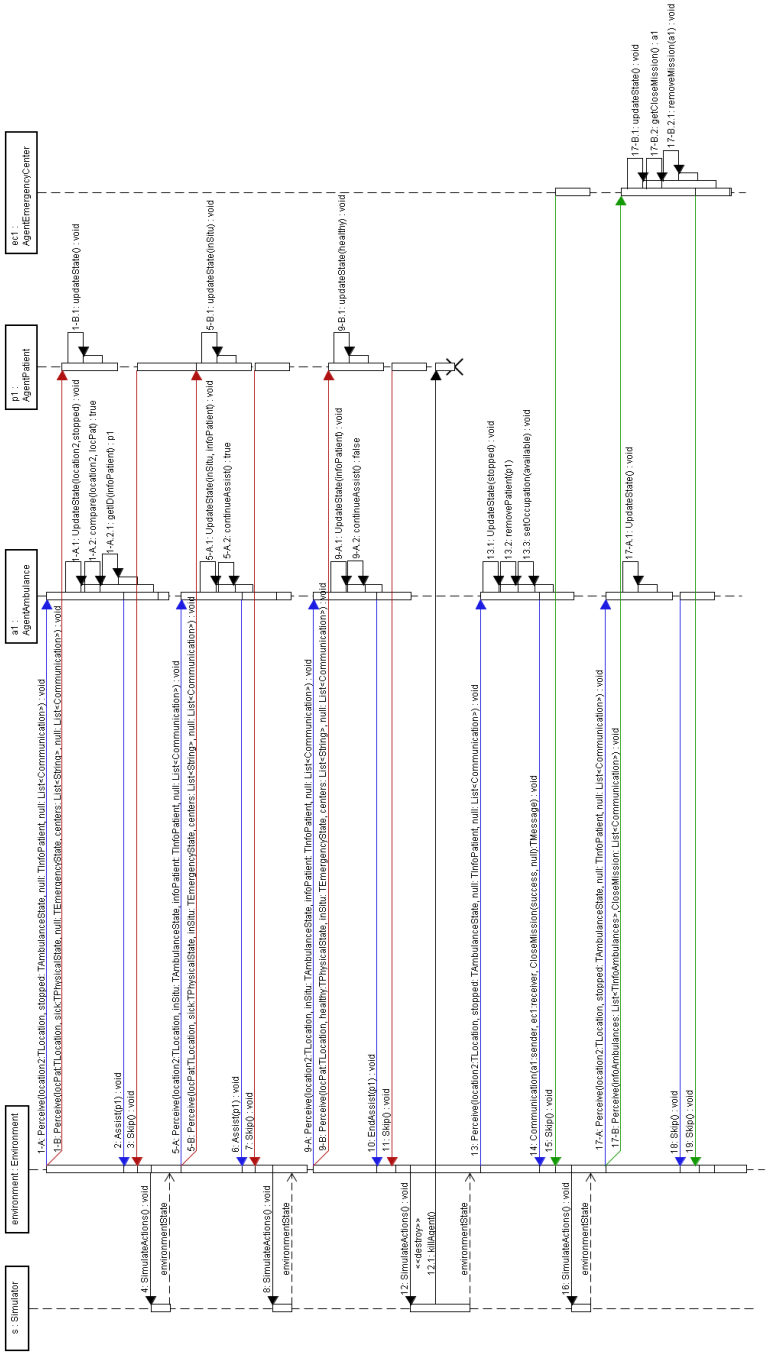


B.1.3. Movimiento de ambulancia hacia paciente

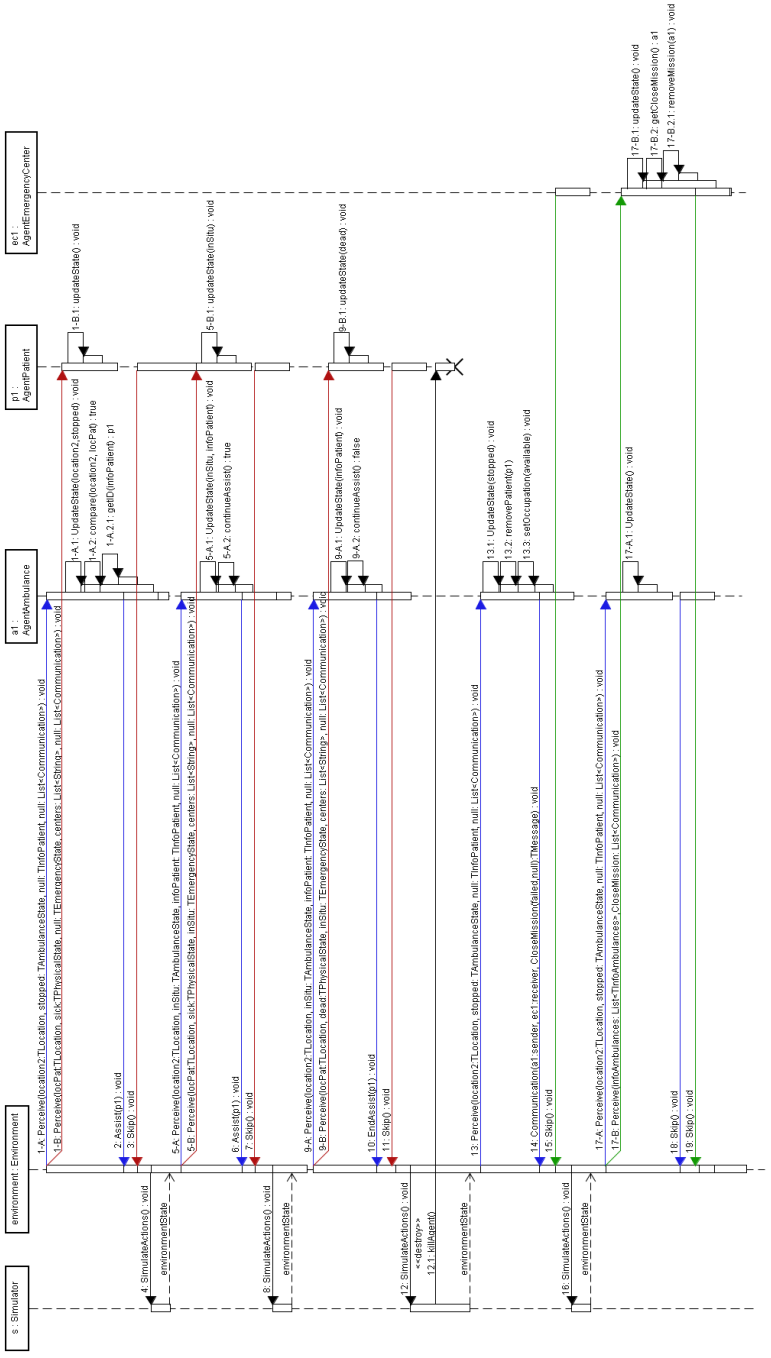


B.1.4. Asistencia de ambulancia a paciente

B.1.4.1. Se cura

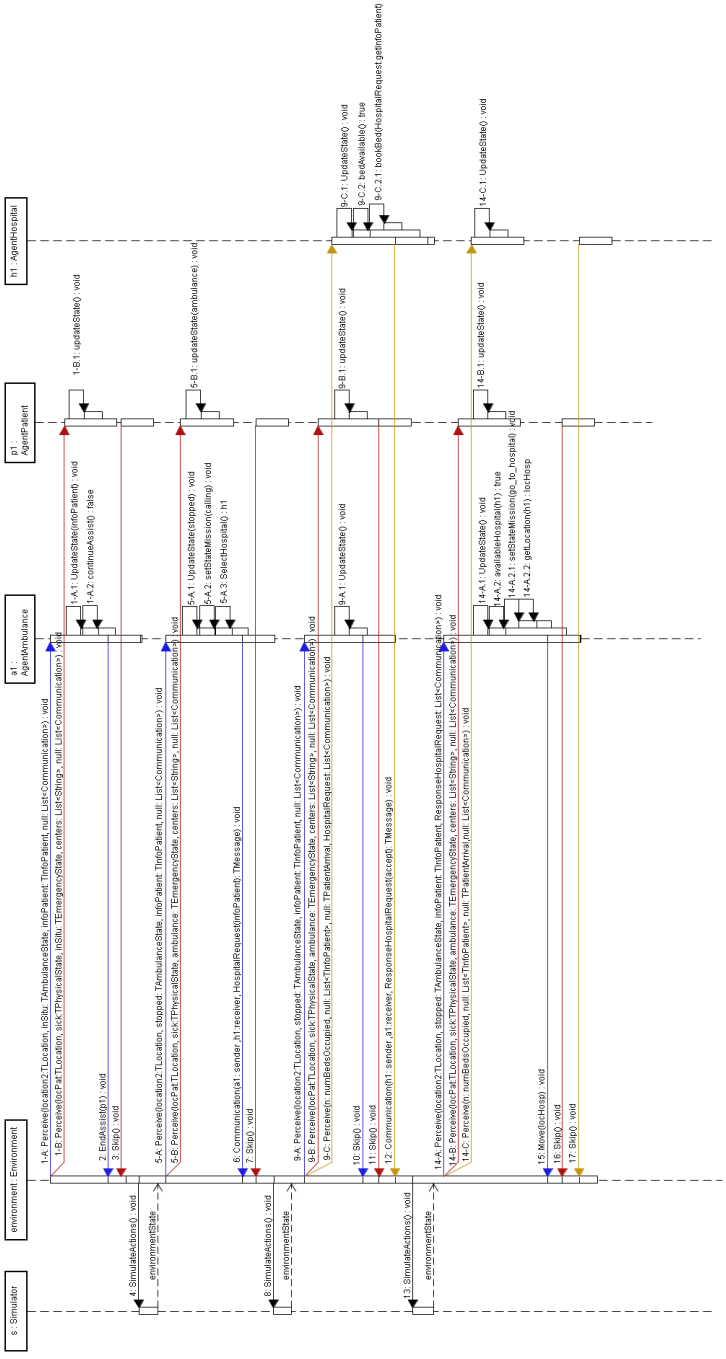


B.1.4.2. Se muere

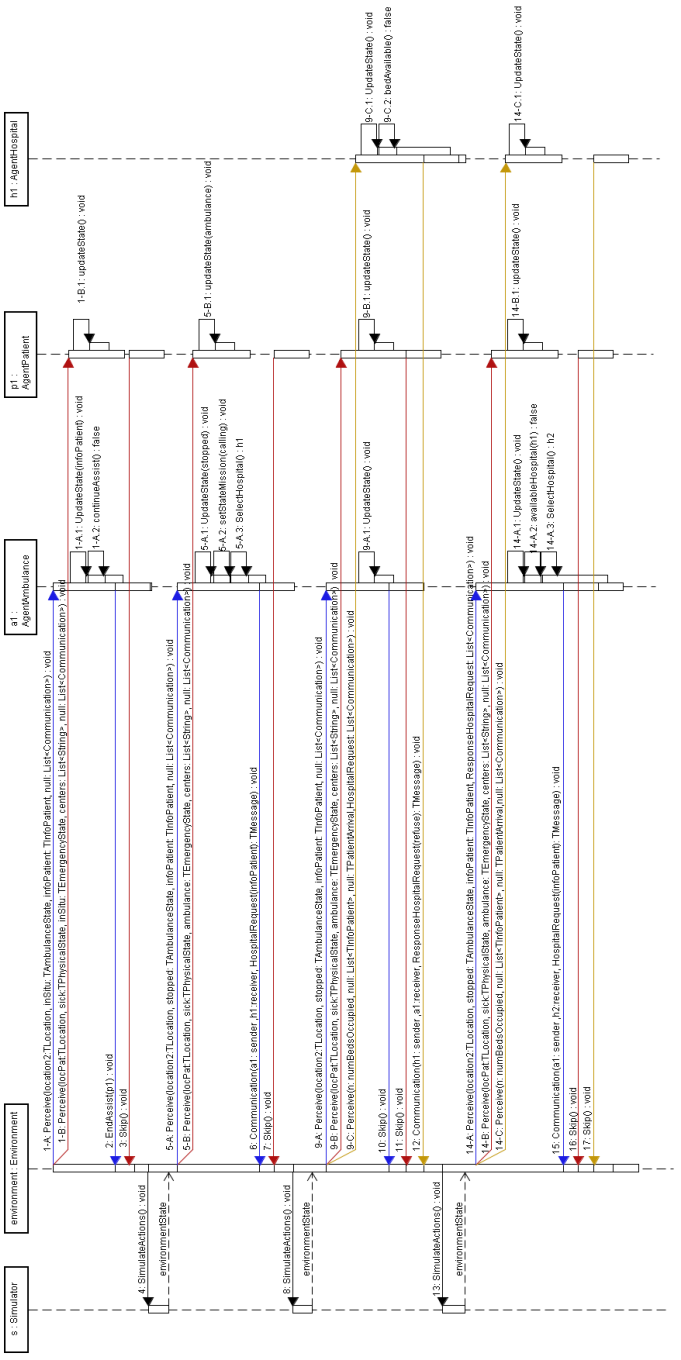


B.1.5. Comunicación de ambulancia con hospital

B.1.5.1. Acepta admision

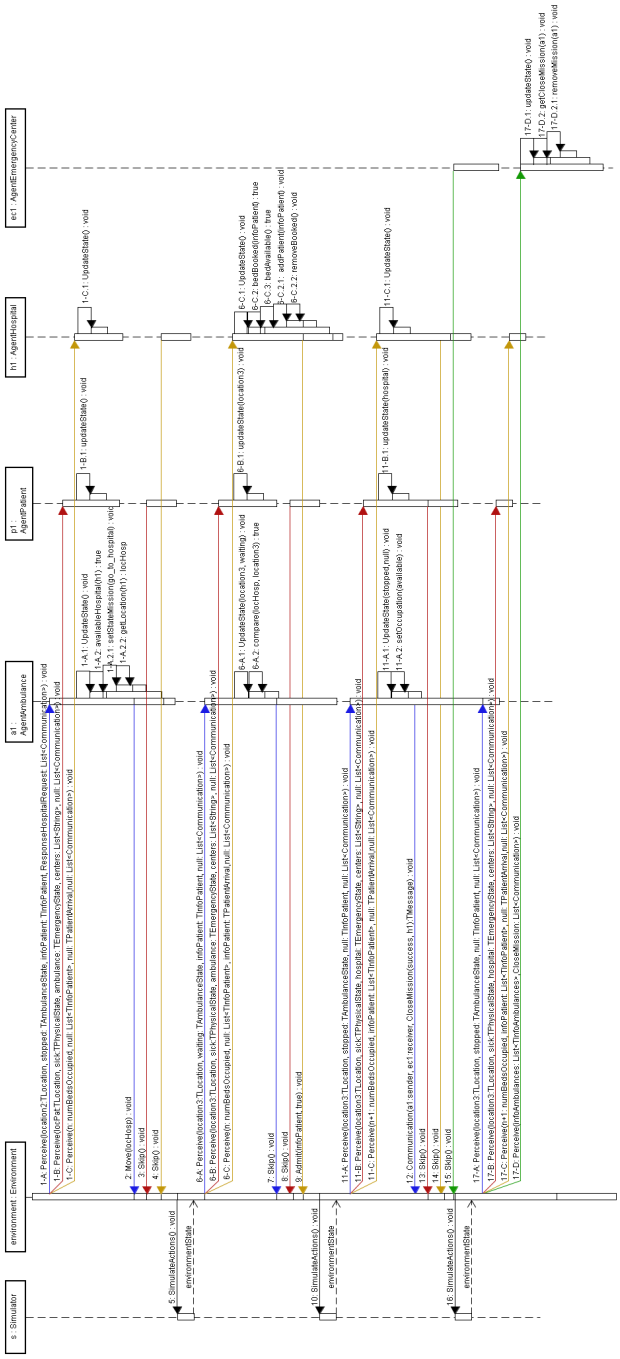


B.1.5.2. Rechaza admision

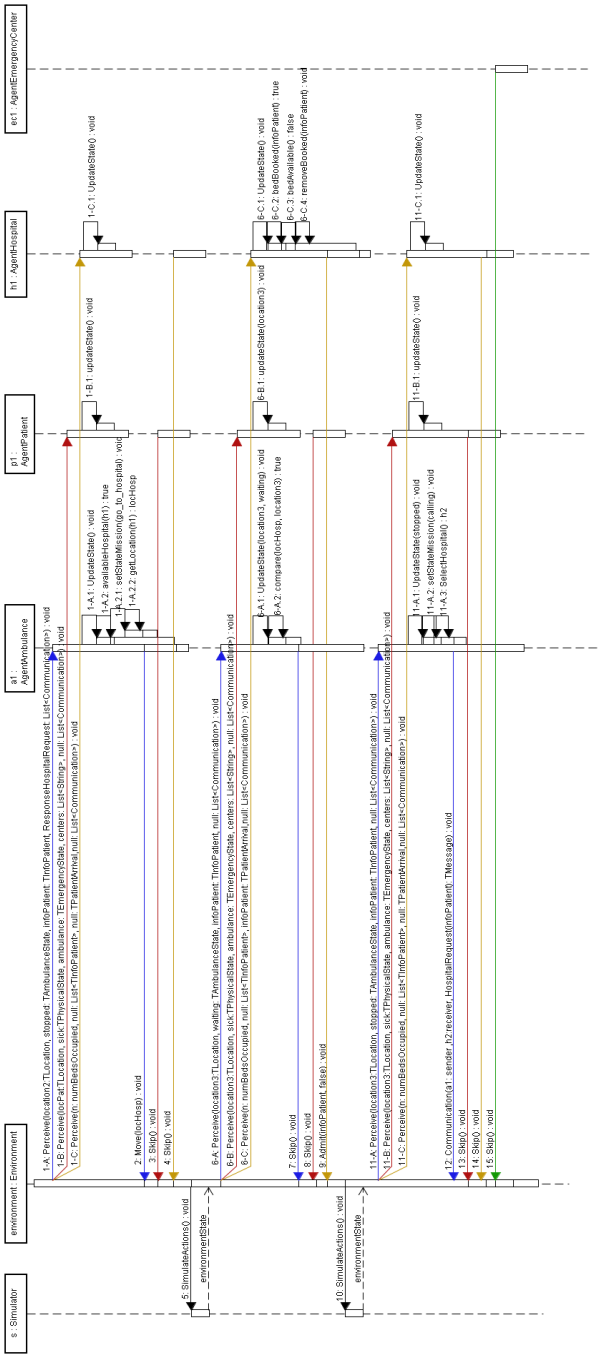


B.1.6. Transporte de un paciente al hospital

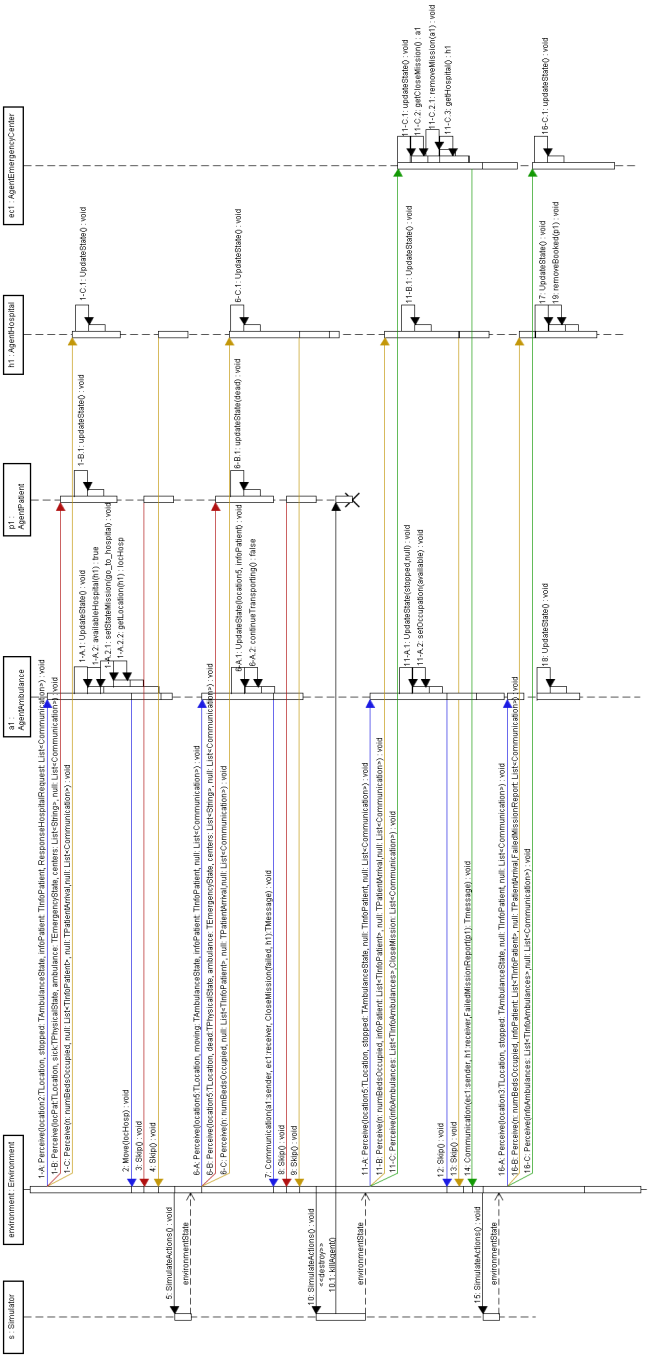
B.1.6.1. El hospital admite



B.1.6.2. El hospital rechaza

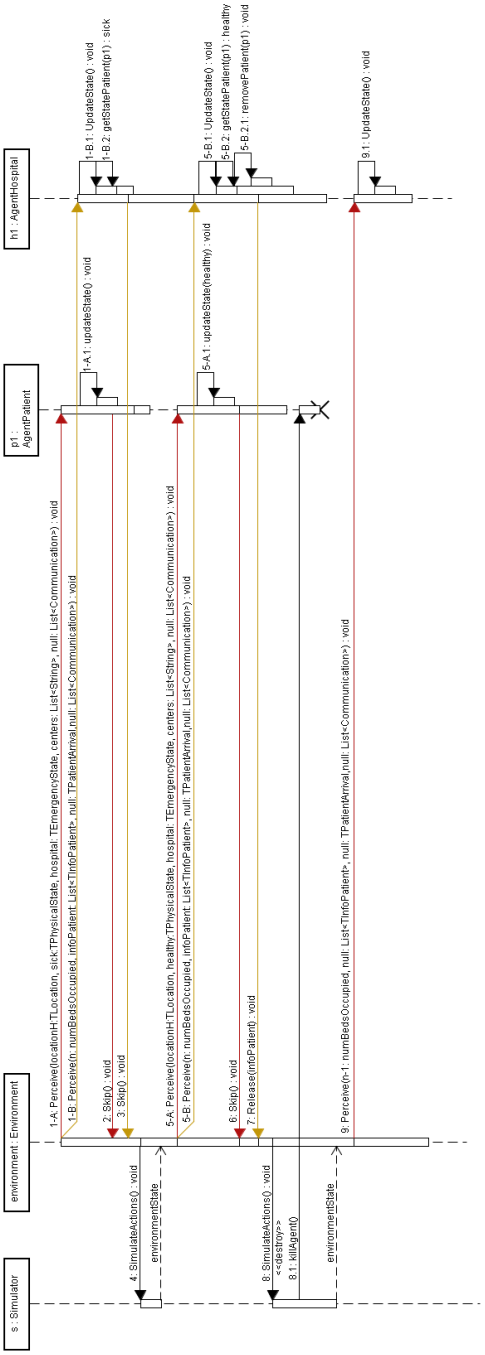


B.1.6.3. El paciente muere durante el transporte

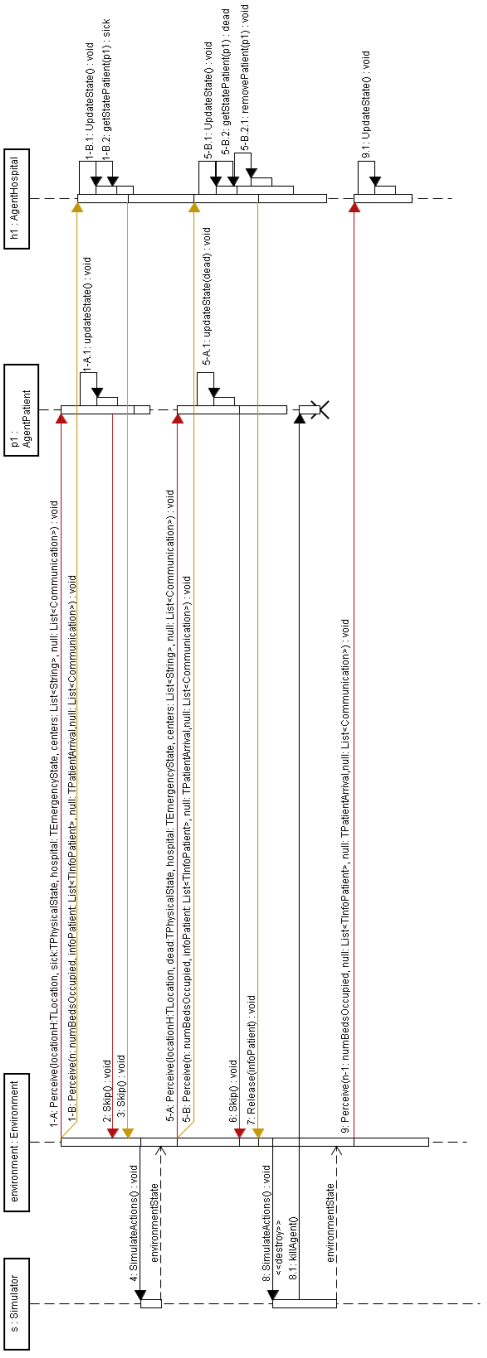


B.1.7. Un paciente se encuentra en el hospital

B.1.7.1. Se muere

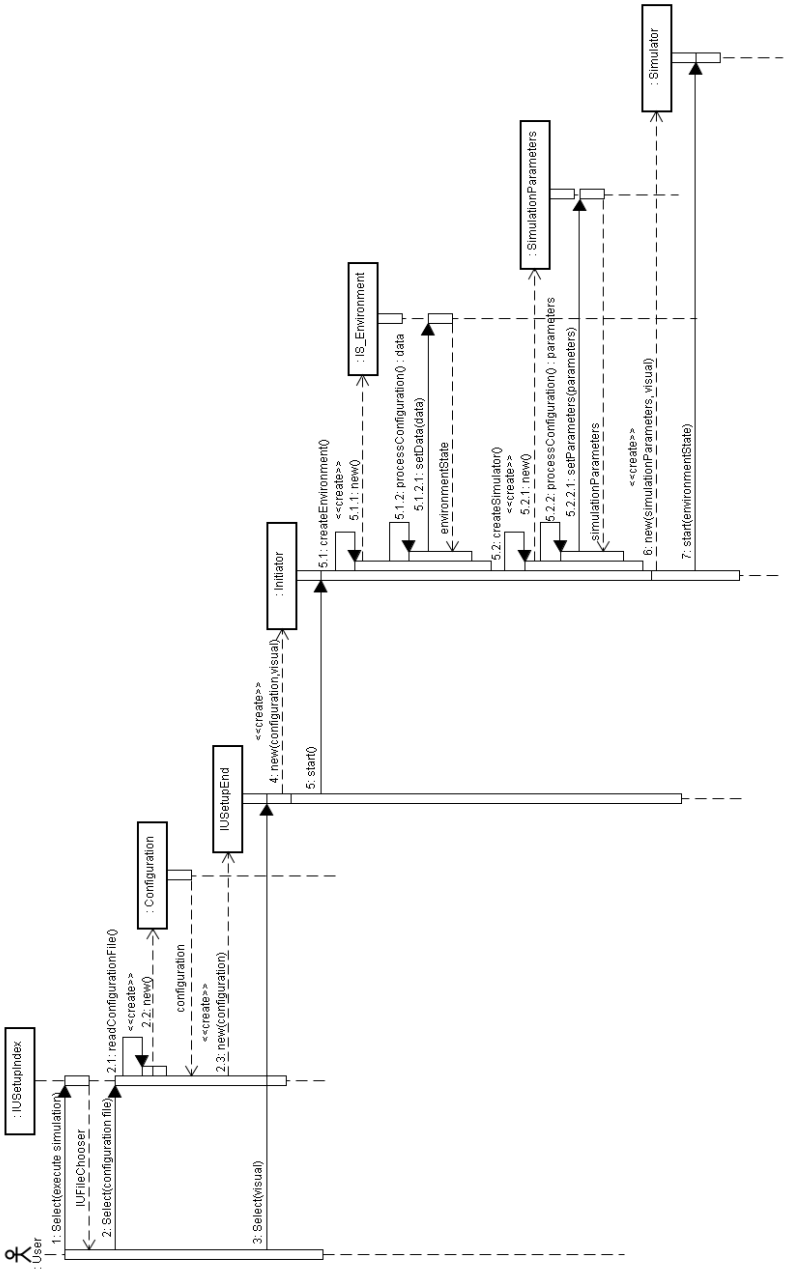


B.1.7.2. Se cura

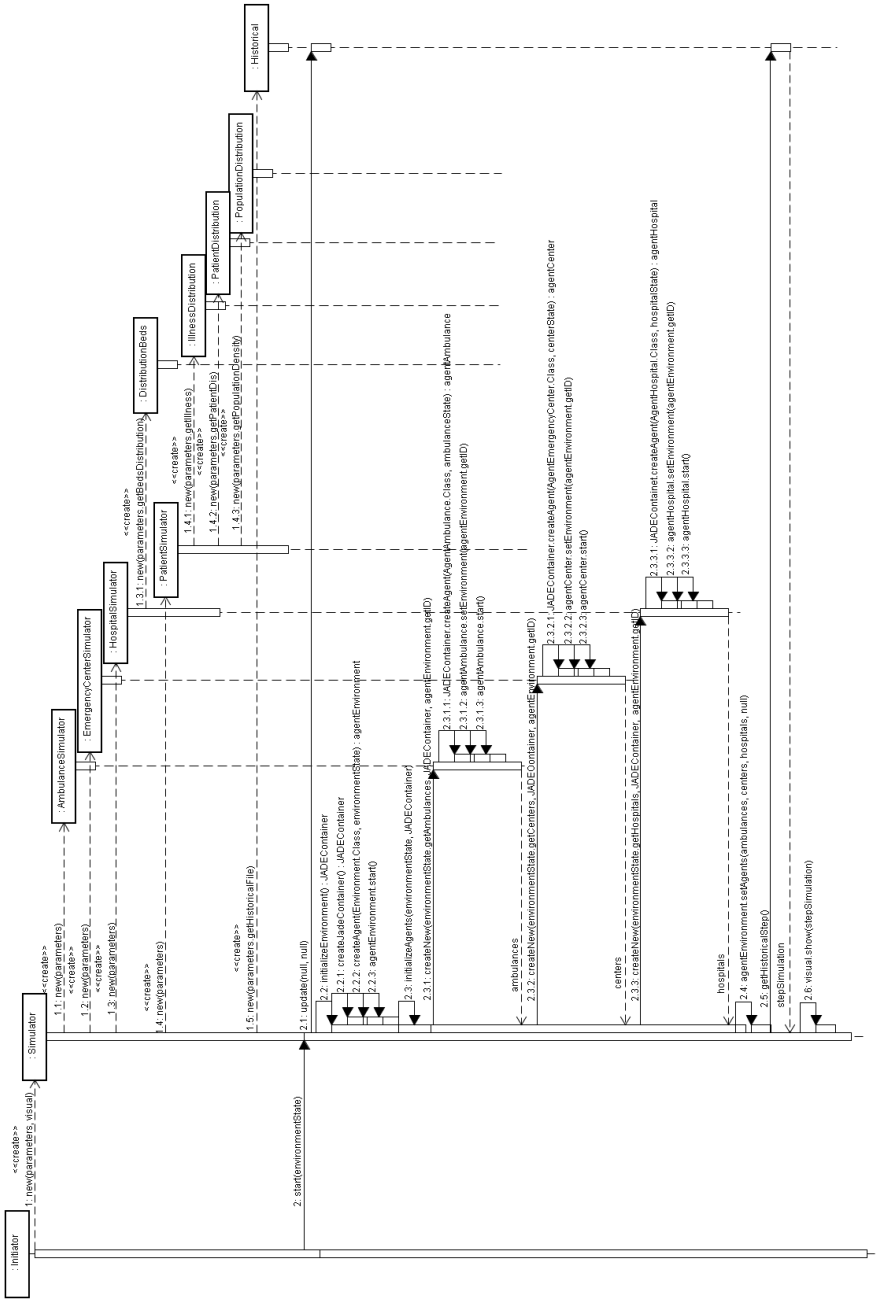

h1 : AgentHospital
p1 : AgentPatient
environment : Environment
s : Simulator
1-B.1: UpdateState() : void
1-B.2: getStatePatient(p1) : sick
5-B.1: UpdateState() : void
5-B.2 : getStatePatient(p1) : dead
5-B.2.1: removePatient(p1) : void
9.1: UpdateState() : void
1-A.1: updateState() :void
5-A.1: updateState(dead) : void
1-A: Perceive(locationH:TLocation, sick:TPhysicalState, hospital: TEmergencyState, <enters: List<String>, null: List<Communication>>) : void
1-B. Perceive(n: numBedsOccupied, infoPatient: List<TInfoPatient>, null: TPatientArrival,null: List<Communication>) : void
2: Skip() : void
3: Skip() : void
5-A. Perceive(locationH:TLocation, dead:TPhysicalState, hospital: TEmergencyState, <enters: List<String>, null: List<Communication>>) : void
5-B. Perceive(n: numBedsOccupied, infoPatient: List<TInfoPatient>, null: TPatientArrival,null: List<Communication>) : void
6: Skip() : void
7: Release(infoPatient) : void
9: Perceive(n-1: numBedsOccupied, null: List<TInfoPatient>, null: TPatientArrival,null: List<Communication>) : void
4: SimulateActions() : void
environmentState
8: SimulateActions() : void
<<delete>>
8.1: killAgent()
environmentState

B.2. Simulador

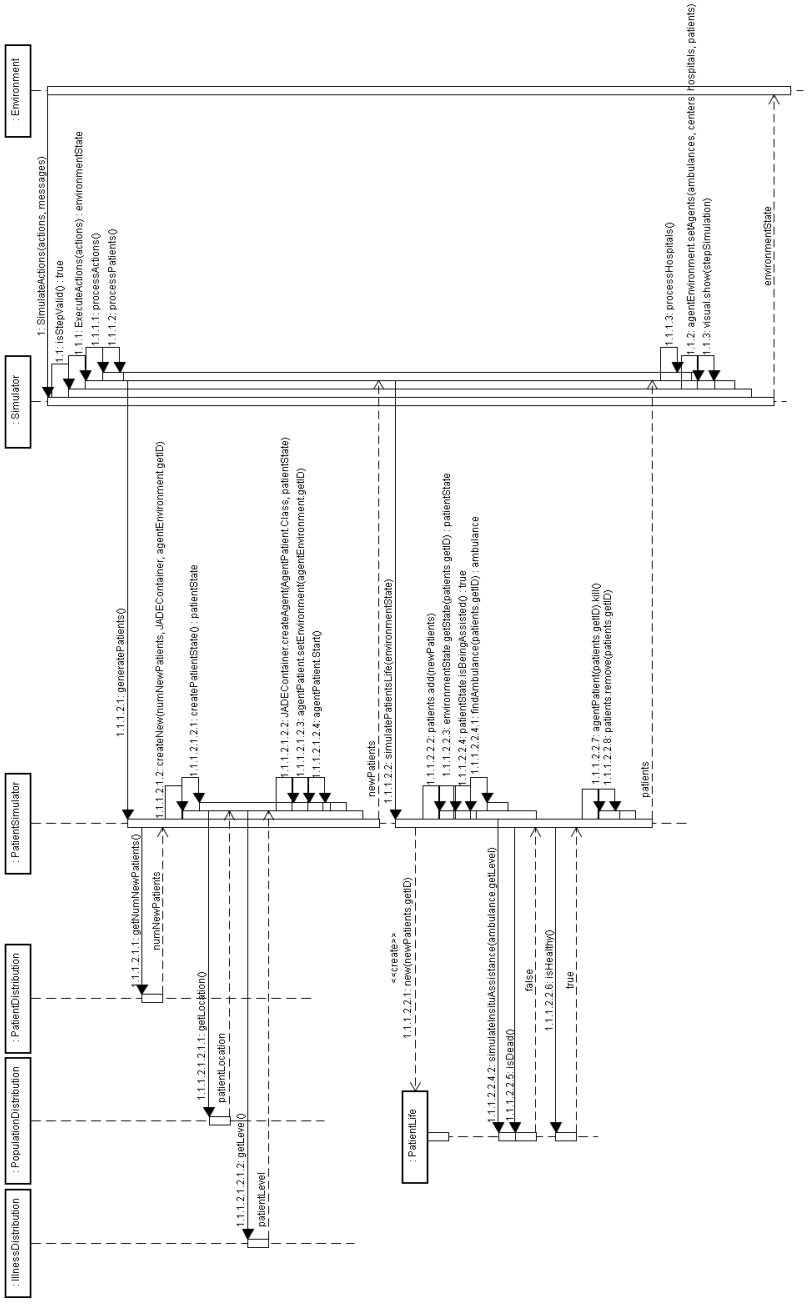
B.2.1. Comienza la simulación de un escenario de emergencias médicas



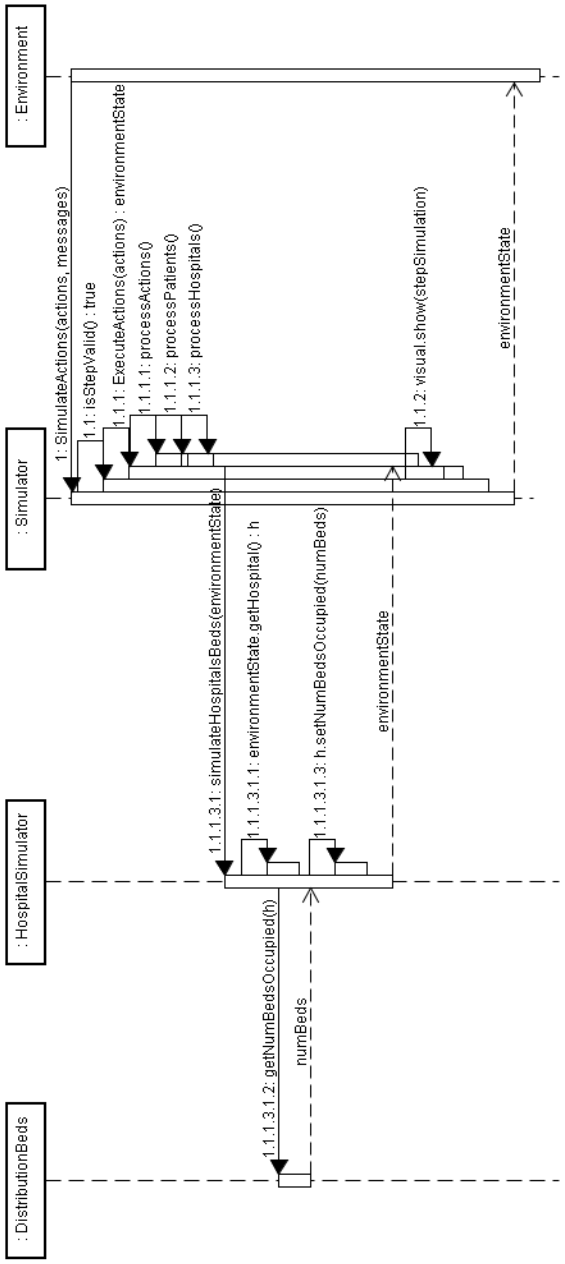
B.2.2. Se crean el entorno y los diferentes agentes que participarán en él en el primer paso de simulación



B.2.3. Se procesan los pacientes del entorno

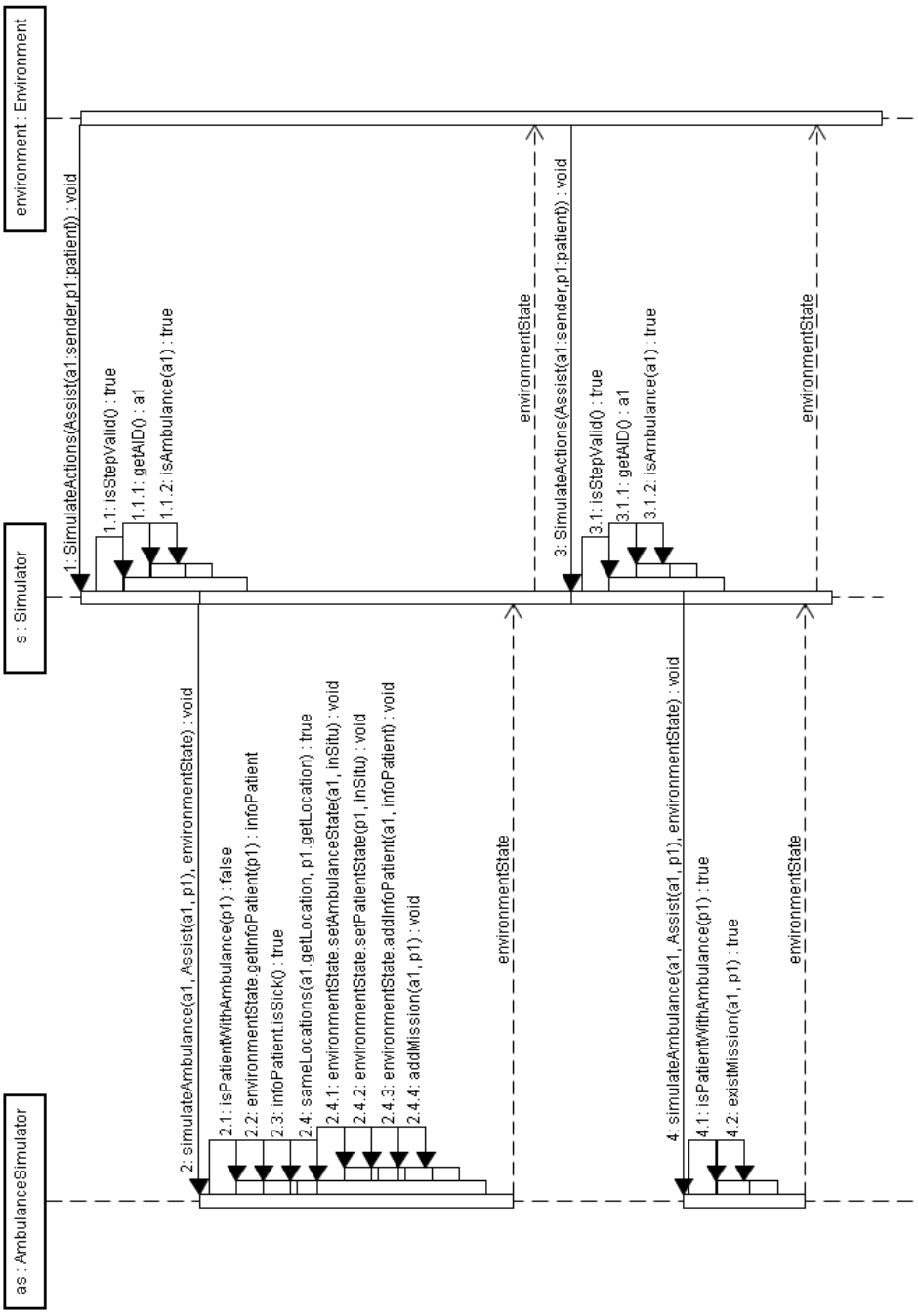


B.2.4. Se procesan los hospitales del entorno

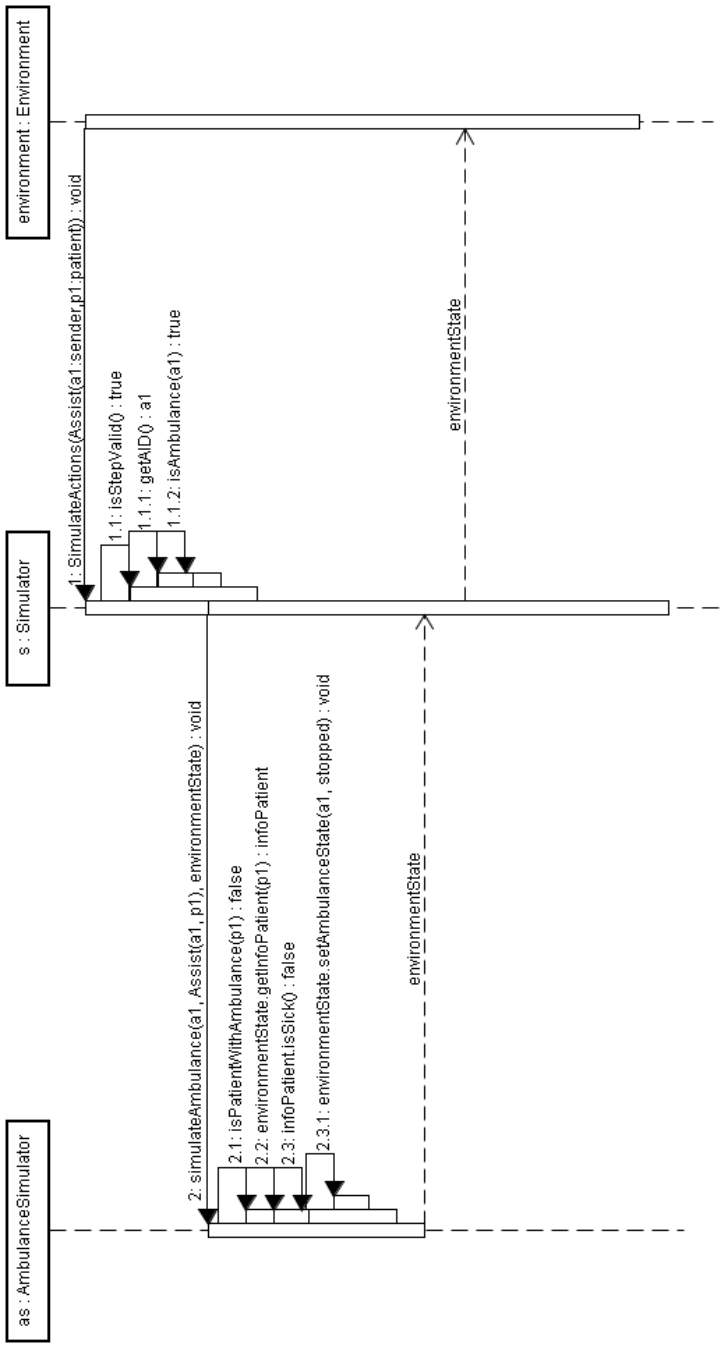


B.2.5. Simulación de cómo asiste una ambulancia a un paciente

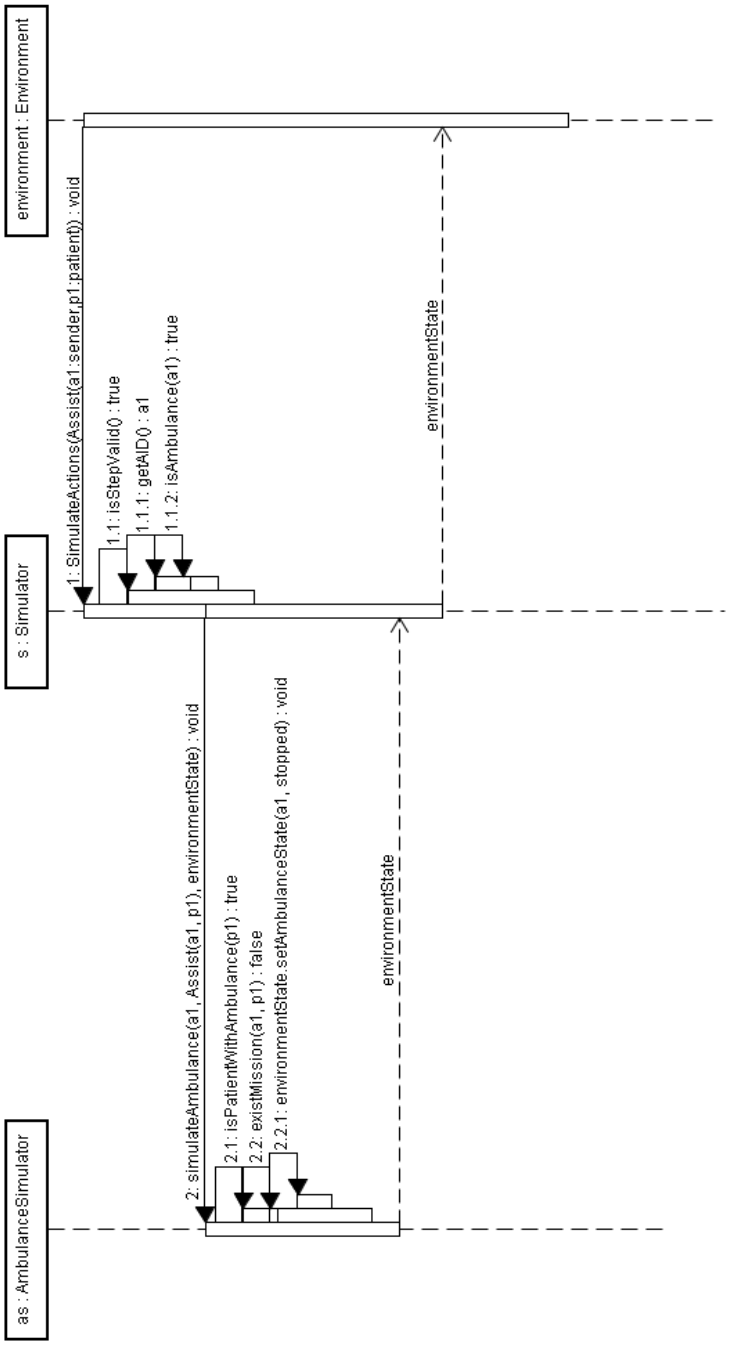
B.2.5.1. El paciente no mejora



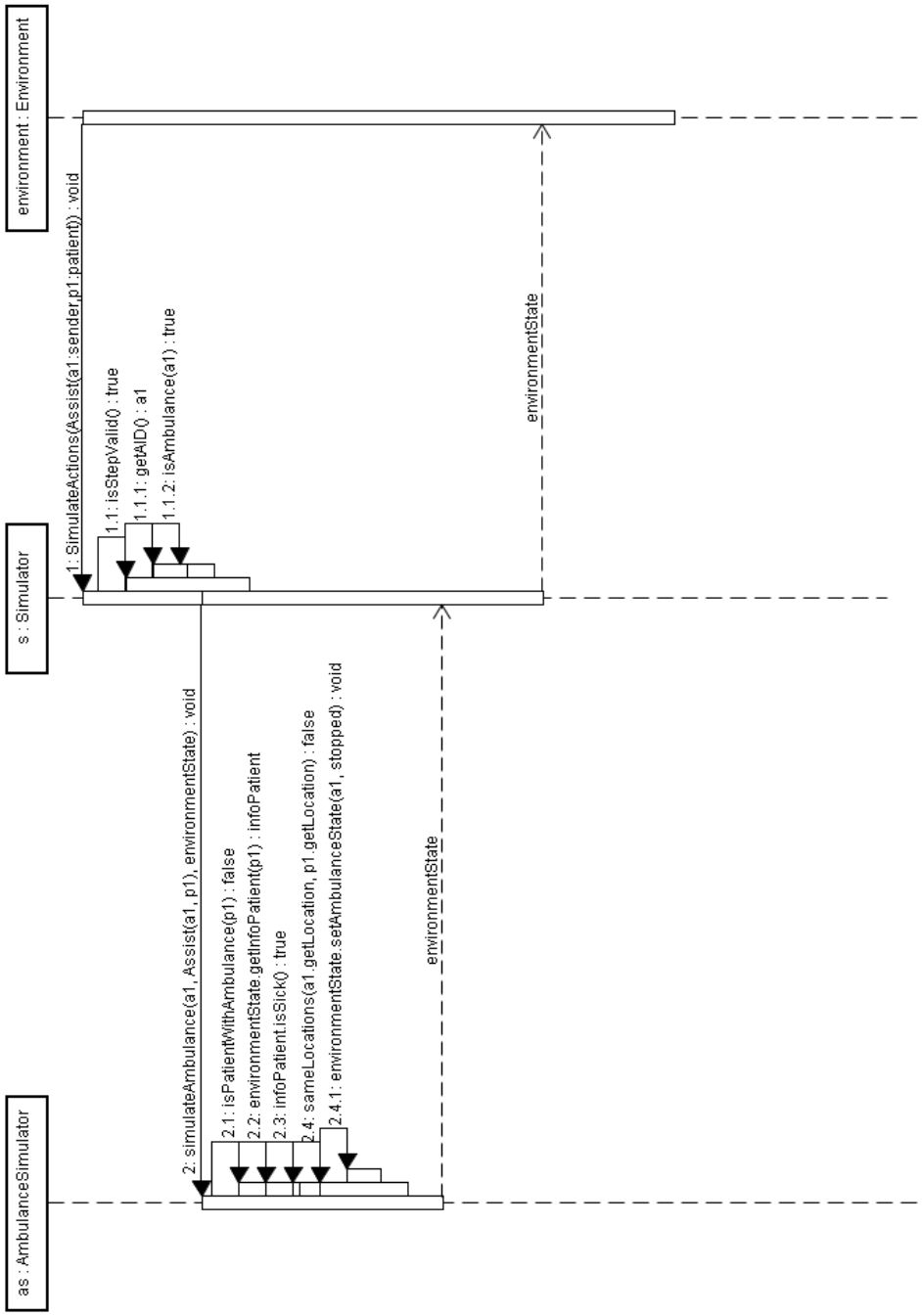
B.2.5.2. El paciente esta muerto o se ha curado cuando la ambulancia le asiste



B.2.5.3. La ambulancia quiere asistir a un paciente que ya está en otra ambulancia

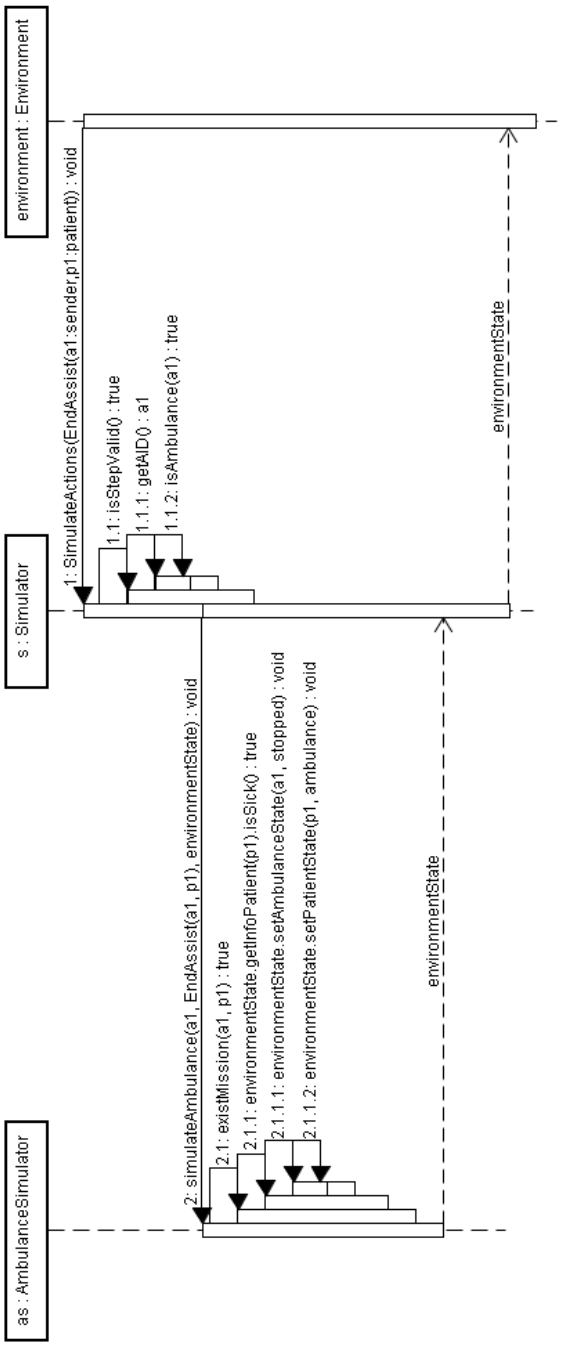


B.2.5.4. La ambulancia quiere asistir a un paciente que no está en la misma localización que ella

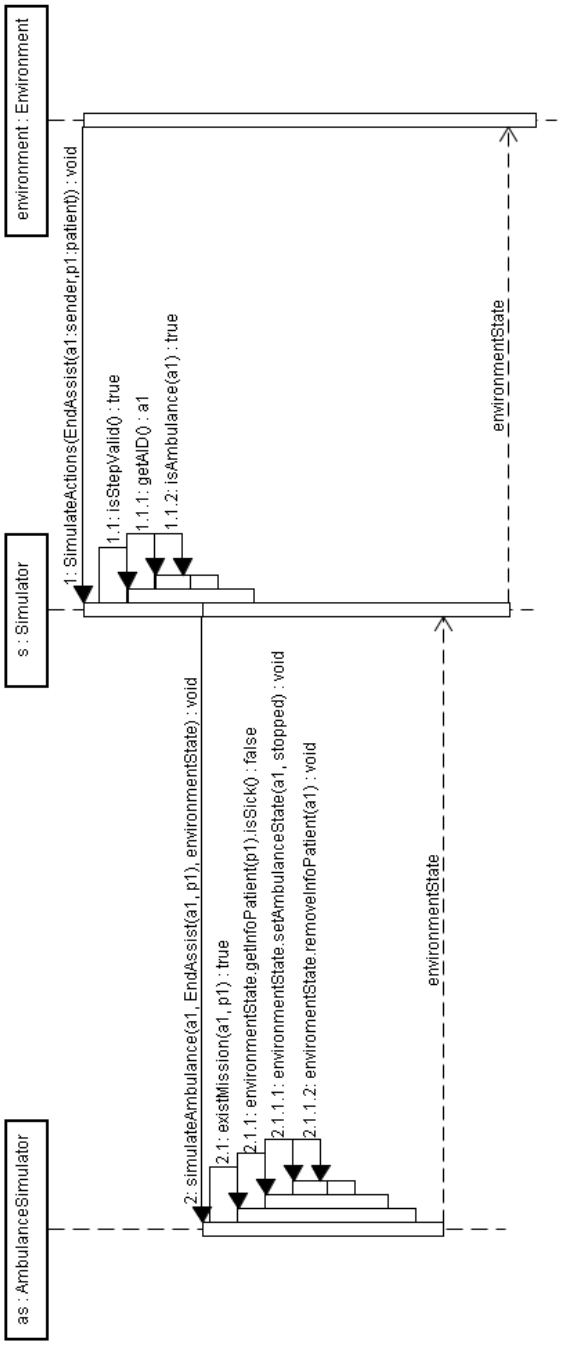


B.2.6. Simulación de cómo finaliza una asistencia una ambulancia

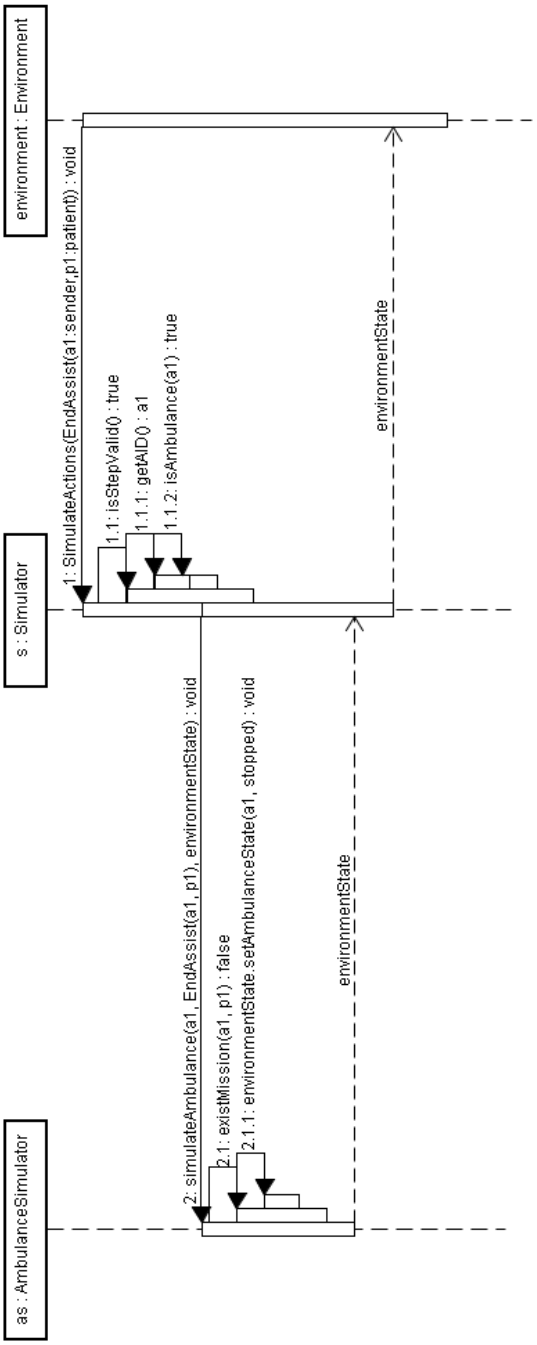
B.2.6.1. Se simula el fin de asistencia y se transporta al paciente a un hospital



B.2.6.2. Fin de asistencia cuando un paciente ha muerto o se ha curado

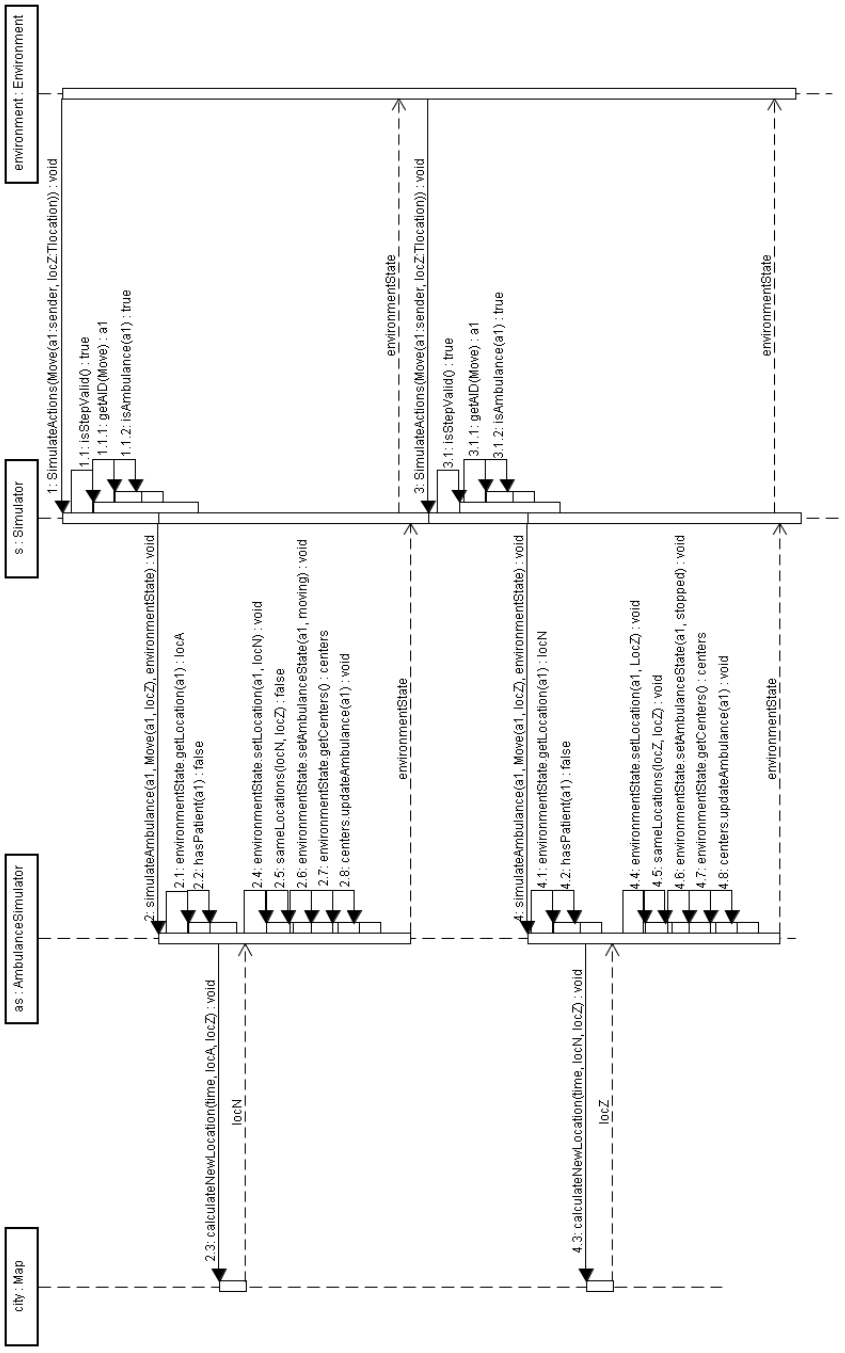


B.2.6.3. La ambulancia quiere terminar la asistencia de un paciente que no tiene

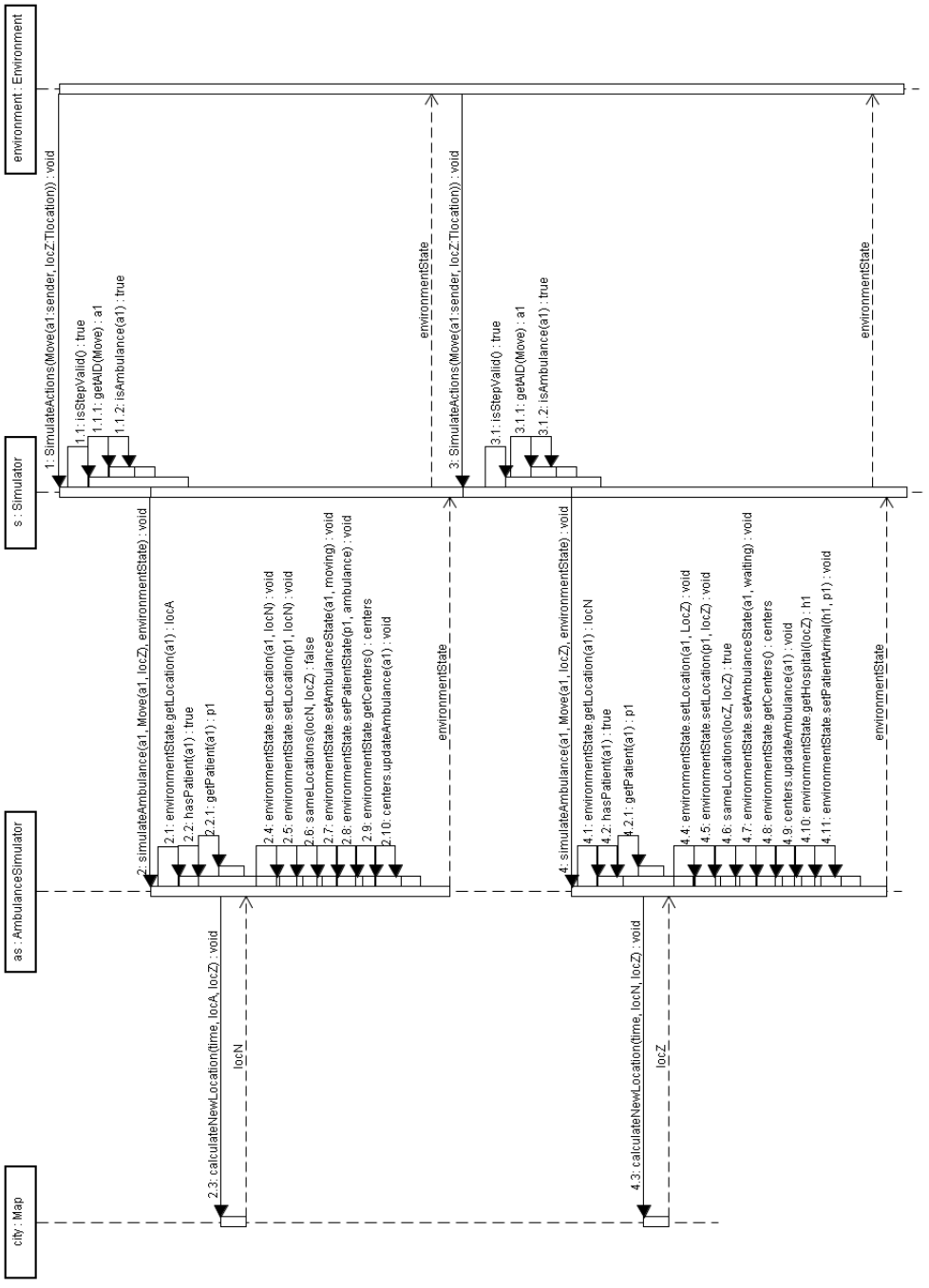


B.2.7. Simulación de los movimientos una ambulancia

B.2.7.1. La ambulancia se mueve hacia un hospital para trasladar a un paciente

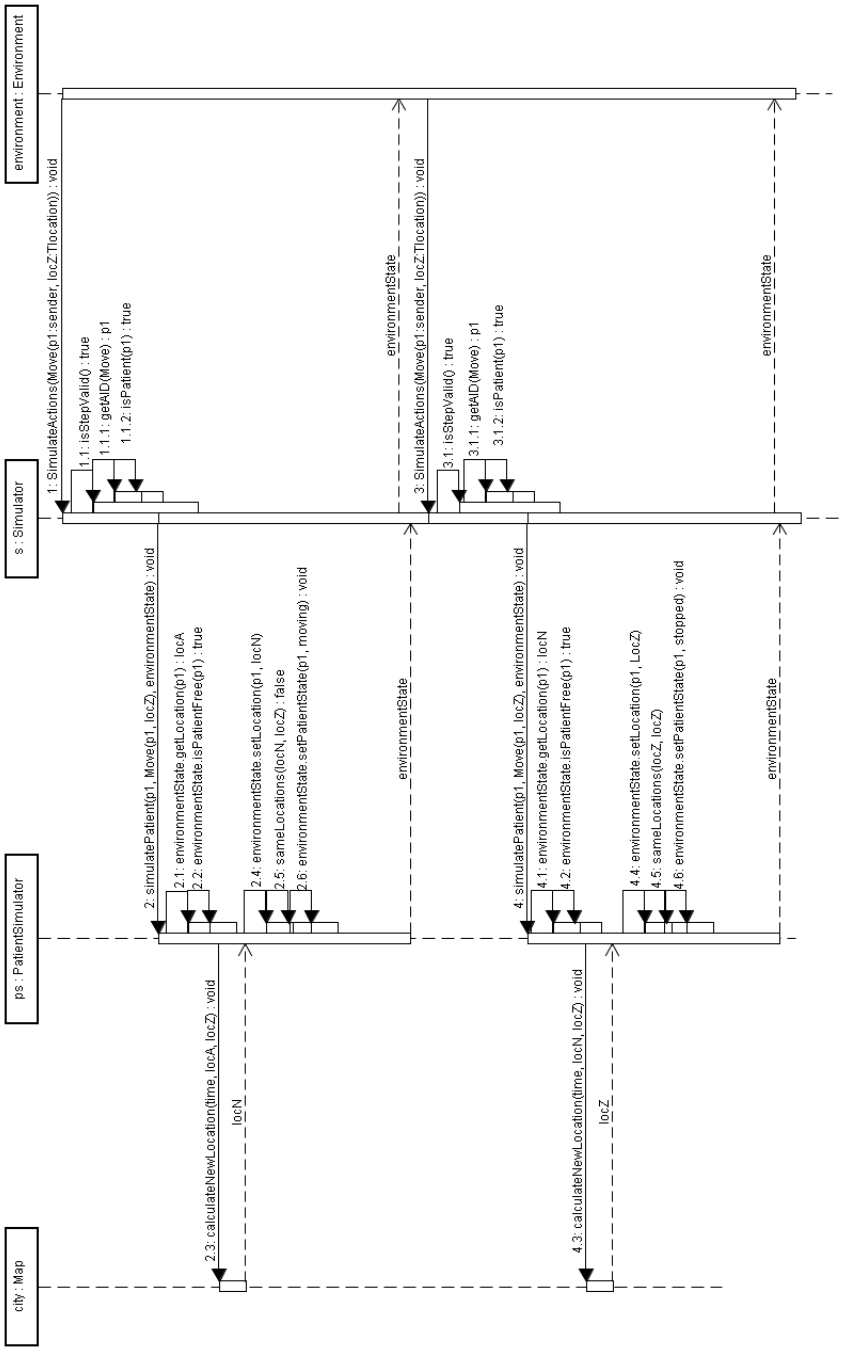


B.2.7.2. La ambulancia se mueve sin tener misión activa

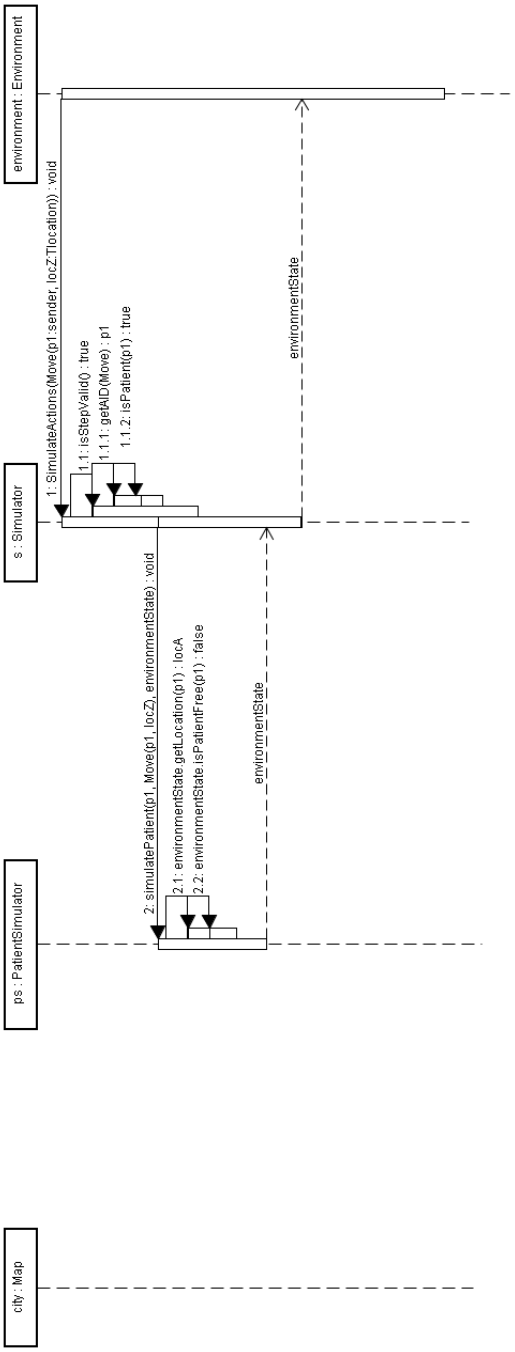


B.2.8. Simulación de los movimientos de una persona

B.2.8.1. La persona se puede mover cuando está sana

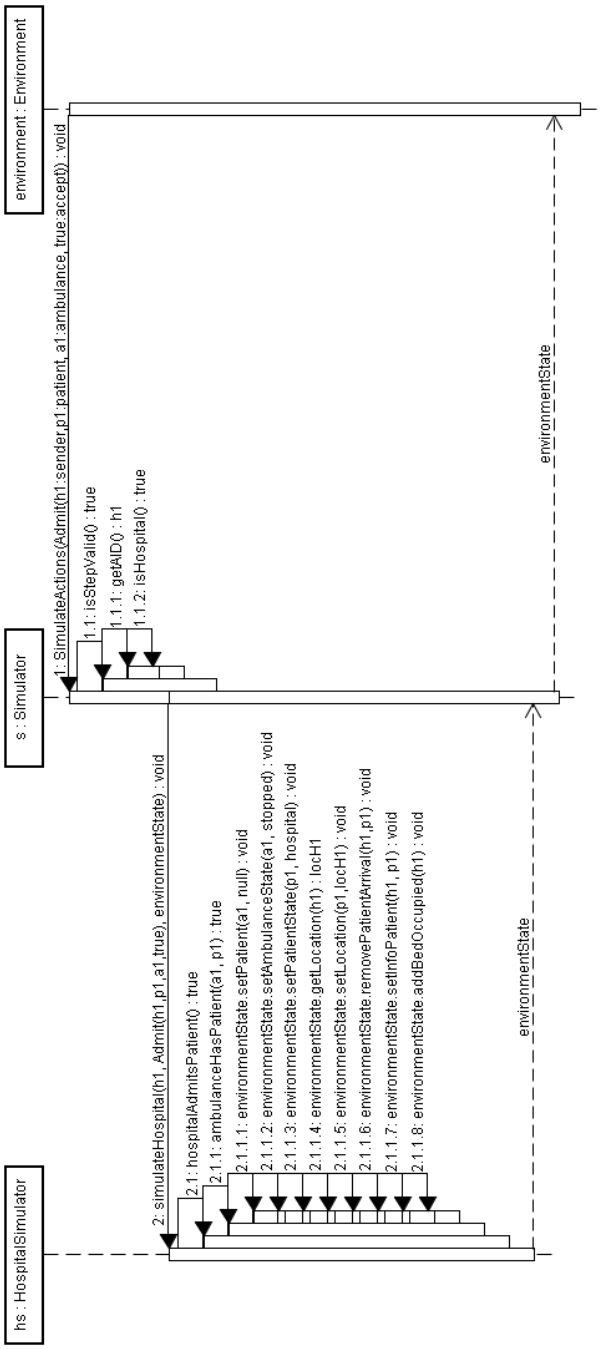


B.2.8.2. La persona no se puede mover porque está enferma, ya sea en hospital, ambulancia o en su casa.

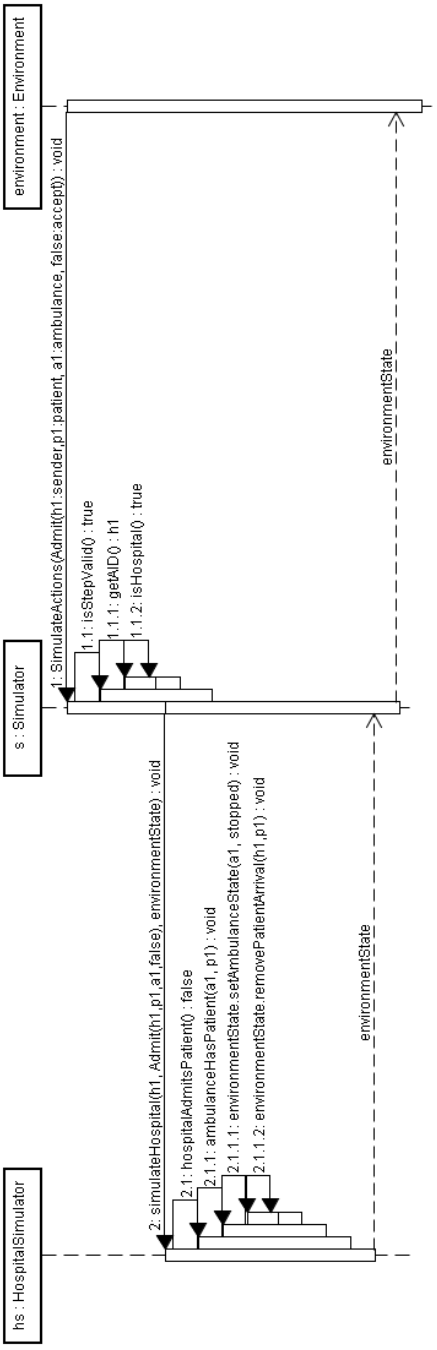


B.2.9. Simulación de la admisión de un paciente en un hospital

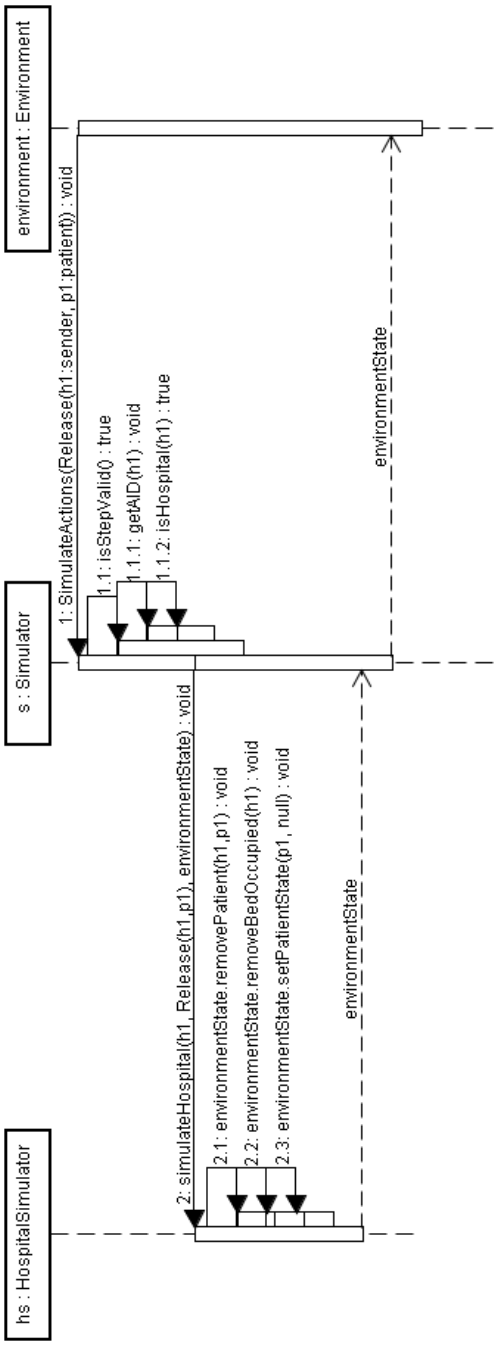
B.2.9.1. El paciente llega al hospital y éste le admite



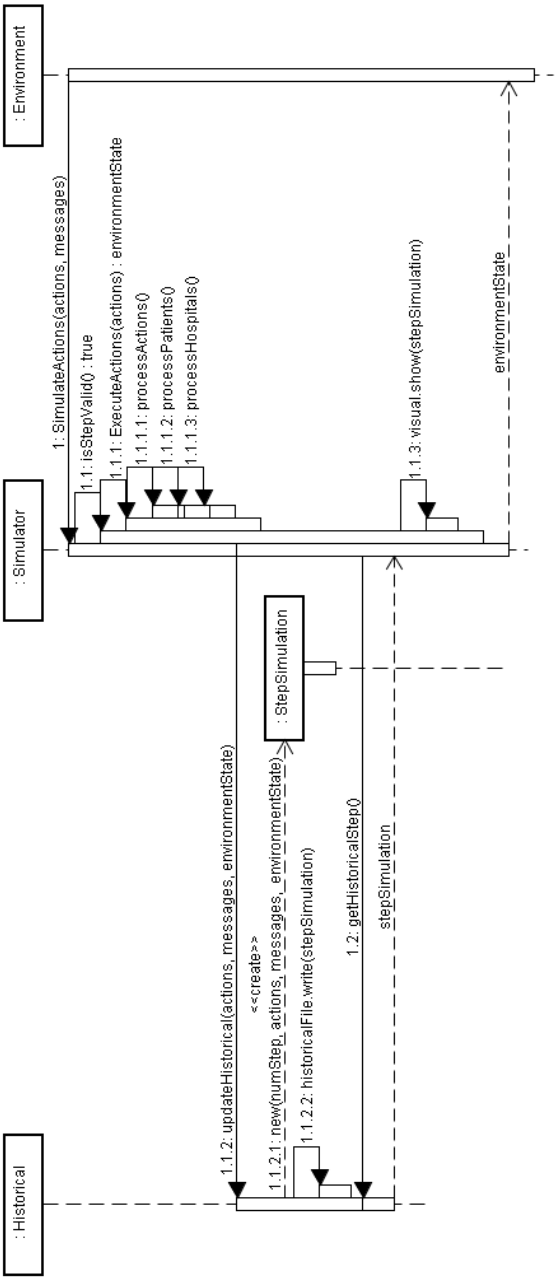
B.2.9.2. El paciente llega al hospital pero éste le rechaza la admisión



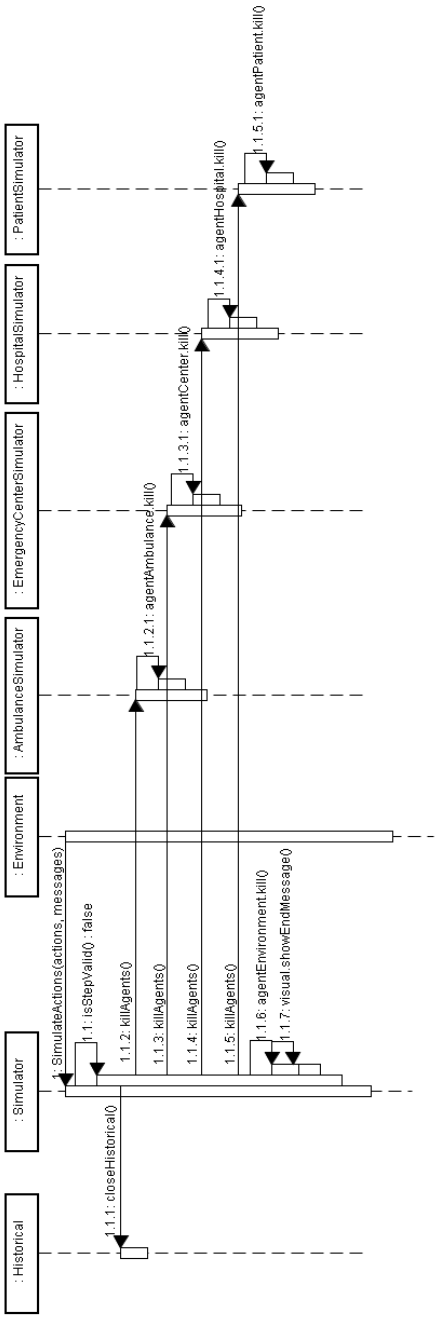
B.2.10. Simulación del alta de un paciente del hospital



B.2.11. Se almacena la información de cada paso de simulación en el fichero histórico

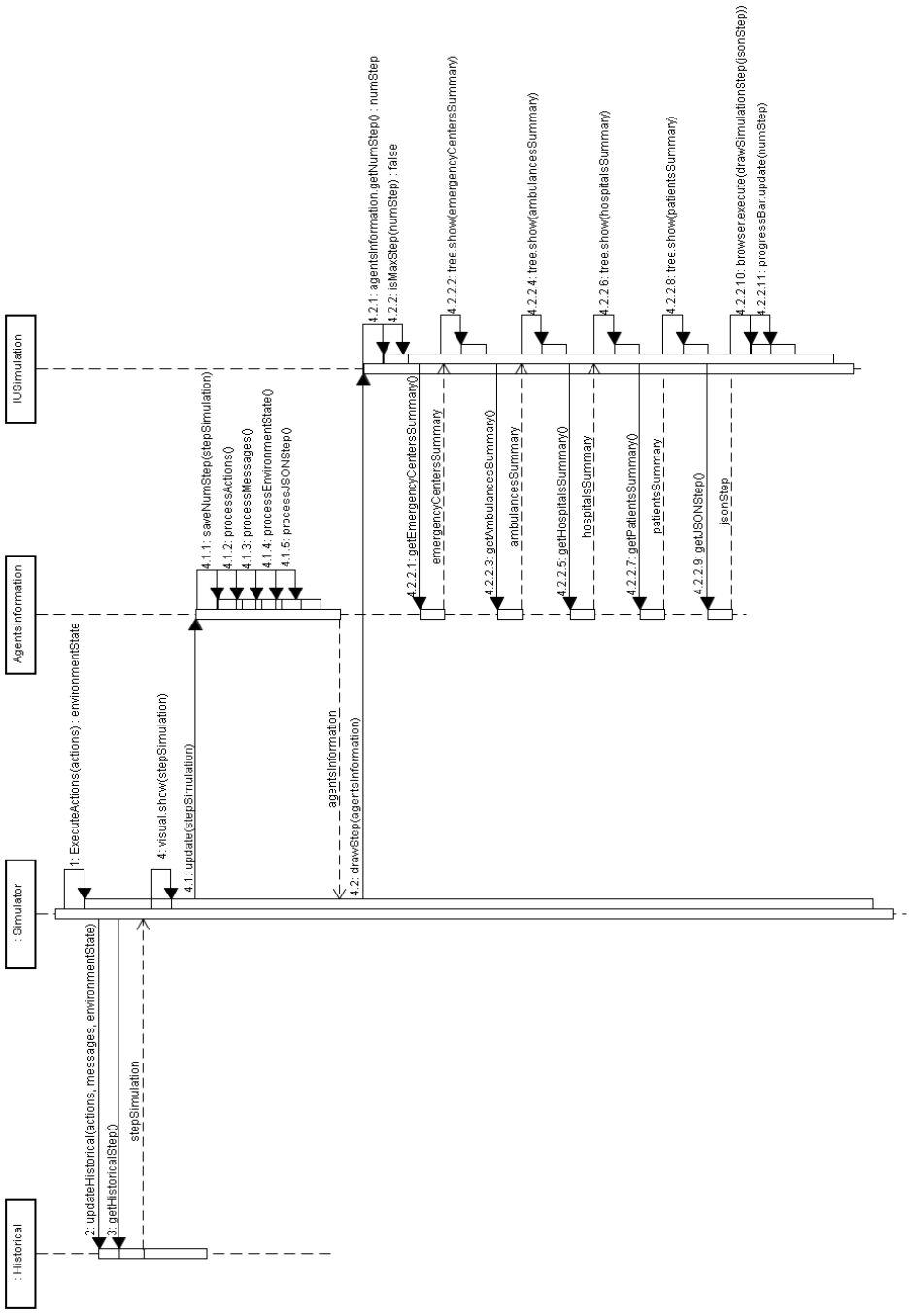


B.2.12. Se finaliza el proceso de simulación



B.3. Visualizador

B.3.1. Visualizar una simulación de forma global



B.3.2. Visualizar la información asociada a un único agente