



Sistemas Informáticos

Curso 2004-2005

Aventura Gráfica Multijugador

Tania Anta Álvarez
Nazaret Sánchez Rodríguez
Santos Hernanz Lillo

Dirigido por:
Prof. Jorge J. Gómez Sanz
Dpto. Sistemas informáticos y programación (SIP)

Facultad de Informática
Universidad Complutense de Madrid

Índice

1.	Resumen	6
1.1.	Resumen	6
1.2.	Abstract.....	6
2.	Creación de historias	7
2.1.	Introducción.....	7
2.2.	Método genérico de definición de historias.....	8
2.2.1.	Componentes de una historia.....	8
2.2.2.	Crear nuevos objetos	9
2.2.3.	Crear nuevos personajes	10
2.2.4.	Acciones	12
2.2.5.	Aumentar la funcionalidad de objetos y personajes.	13
2.2.6.	Modificar el mapa.....	16
2.2.7.	Añadir objetos iniciales en el inventario de un jugador	18
2.2.8.	Representación grafica	18
2.3.	Aplicación a un caso práctico: 10% de Azar.....	19
2.3.1.	Descripción global de la historia	19
2.3.2.	Definición de personajes y objetos.....	19
2.3.3.	Flujo de sucesos.....	22
2.3.4.	Codificación de objetos y personajes	24
2.3.5.	Remodelación del mapa y ubicación de los objetos y personajes en él .	39
2.3.6.	Definir los objetos del inventario por defecto	42
2.3.7.	Representación grafica de los objetos y personajes.....	42
2.3.8.	Ensamblado del codigo generado.....	44
3.	Pruebas XML.....	46
3.1.	Introducción.....	46
3.2.	Definición del lenguaje XML.....	46
3.2.1.	Especificación del lenguaje con XML Schema.....	46
3.2.2.	Pruebas.xsd.....	46
3.2.3.	Plantilla para pruebas	48
3.3.	Procesamiento de documentos XML.....	48
3.3.1.	DOM.....	48
3.3.2.	Requisitos	49
3.4.	Diseño del paquete de pruebas	49
3.4.1.	Diagrama de casos de uso.....	49
3.4.2.	Diagrama de actividades.....	50
3.4.3.	Diagrama de secuencia	51
3.4.4.	Paquetes	51
3.5.	Arquitectura del paquete.....	52
3.5.1.	Diagrama de clases	52
3.5.2.	PruebasParser	53
3.5.3.	PruebaCliente	54
3.5.4.	DatosPrueba.....	55
3.5.5.	Interfaz.....	56
3.5.6.	DatosConexión	56
3.5.7.	DatosCreación	57
3.5.8.	Espera	58
3.5.9.	Filtro	58

3.5.10.	Acción.....	59
3.6.	Tipos de pruebas.....	60
3.7.	Tutorial de pruebas.....	60
3.7.1.	Componentes involucrados.....	60
3.7.2.	Acciones a efectuar.....	63
3.7.3.	Escribir la prueba.....	63
3.8.	Aplicación a un caso práctico: 10% de azar.....	64
3.8.1.	Puertas implicadas.....	64
3.8.2.	Objetos implicados.....	64
3.8.3.	Personajes no jugadores implicados.....	64
3.8.4.	Acciones involucradas.....	64
3.8.5.	Escribir la prueba de la aventura.....	65
3.9.	Ejecutando la aplicación.....	65
3.10.	Integración con AGM.....	66
3.10.1.	ClienteAdministrador.....	66
3.10.2.	ClienteJugador.....	66
3.11.	Mensajes del servidor.....	68
3.12.	Probando AGM.....	68
3.13.	Conclusiones.....	70
4.	Animación en la AGM.....	73
4.1.	Introducción.....	73
4.1.1.	Requisitos.....	74
4.2.	Pruebas de investigación.....	74
4.2.1.	Ideas teóricas. Evolución de las mismas.....	74
4.2.2.	Estructura del programa de pruebas.....	80
4.2.3.	Conclusiones.....	106
4.3.	Gamelib.....	107
4.3.1.	nu.bugbase.gamelib.sprite.....	108
4.3.2.	nu.bugbase.gamelib.utils.....	110
4.3.3.	Conclusiones.....	111
4.4.	Ficheros de Configuración.....	112
4.4.1.	Estructura de un XML de configuración.....	112
4.4.2.	Ampliaciones a realizar sobre los XML.....	115
4.5.	Integración con AGM.....	115
4.6.	Cómo crear una animación para un personaje en AGM.....	116
4.6.1.	Ejemplo práctico. Definición de la animación para uno de los “skins” de un PJ.....	117
4.7.	Fuentes de sprites.....	123
5.	Troubleshooting.....	126
5.1.	Objetos en el inventario inicial.....	126
5.2.	Cierre anómalo del cliente con el botón aspa.....	126
5.3.	Al ejecutar la acción USAR de un objeto no se recibe respuesta y se pierde la interacción con dicho objeto.....	126
5.4.	El servidor jboss no consigue conectar y cae en un sin fin de intentos de reconexión.....	127
5.5.	Algunas acciones se repiten sin ninguna razón o patrón aparente.....	127
5.6.	Al realizar una conversación algunas frases no aparecen.....	127
5.7.	Desaparecen los objetos de un inventario.....	128
5.8.	Inclusión de librerías.....	128
5.9.	Solución a error del JBoss.....	128

5.10.	Carga de la ip	128
5.11.	Conexión al juego.....	129
5.12.	Null Pointer Exception	129
5.13.	Puerto ocupado	129
5.14.	Out of memory	129
5.15.	Error en carga de imagenes	130
5.16.	Error puntual en lectura de XML	130
5.17.	Excepcion debida a la ordenación del codigo	130
6.	Bibliografía.....	132
7.	Palabras clave	133
8.	Cesion de derechos	135

1. Resumen

1.1. *Resumen*

El presente proyecto es una continuación de uno desarrollado en el año 2003-2004. Se trata de una aventura gráfica multi-jugador en entornos distribuidos. Para su desarrollo se ha empleado la tecnología J2EE y el entorno de desarrollo Eclipse.

El juego consiste en aprobar de manera fraudulenta las asignaturas de la carrera de Ingeniería en Informática, ya sea en solitario o con la colaboración de otros jugadores.

En la presente documentación se registra nuestro trabajo realizado en este curso académico que se ha centrado en tres líneas básicas. De este modo, en el primer capítulo se relata como crear aventuras en el juego y las posibilidades de la aplicación en este aspecto; en el segundo, se trata el desarrollo de una aplicación para realizar pruebas de la aplicación usando XML y su tutorial para crear nuevas pruebas; en el tercero, se muestra el trabajo realizado para la mejora de la estética del juego, gracias a la inclusión de animaciones y la metodología para configurar animaciones.

1.2. *Abstract*

The current project resumes a former one, developed along the last course, 2003-2004. It consists of a multiplayer graphic adventure, in distributed environments. J2EE technology has been used for its implementation, and Eclipse was the chosen IDE. The goal of the game is to pass, by not quite ethic means, the subjects conforming the Computing degree, either on our own, or by cooperating with other players.

This document records our work, made during this academic year, focused on three main lines. Thus, the first chapter deals with creating a new adventure for the game, as well as explaining the possibilities the application offers in this aspect. The second one tells of the development of a program used to test the main application, using XML, and a tutorial showing how to develop some new proofs. Last, the third chapter shows the work made in order to improve the game's aesthetics, thanks to the inclusion of animations, as well as a guideline for setting and configuring these animations.

2. Creación de historias

2.1. Introducción

Este documento tiene la finalidad de documentar e instruir en el proceso de implementación de una historia con el framework AGM.

El tutorial consta de dos partes principales: El capítulo 2.2 donde se dan las nociones básicas y teóricas para entender el manejo de las historias de cara a la creación de una nueva y el capítulo 2.3 en el que se implementa una historia siguiendo los pasos contemplados en el capítulo 2.2.

En el capítulo 2.2 se encuentran las plantillas utilizadas para crear los objetos y personajes no jugadores (los personajes que no se corresponden a los usuarios) y las directrices para modificarlos en función del papel que queremos que representen. A su vez se explica la distribución del mapa actual de la AGM y como modificarlo para ampliarlo y ubicar los objetos y personajes que constituyen las historias.

En el capítulo 2.3 se implementa la historia *10% de azar* que presenta a los personajes *Secretario* y *Profesor SOS* y los objetos *Caja de Carnés I*, *Carné I*, *Carné J*, *Taquilla*, *Llave taquilla* y *Aprobado SOS*.

Al final de la lectura de este documento el usuario podrá implementar sus propias historias con facilidad.

2.2. Método genérico de definición de historias

La definición de historias con el framework AGM, se realiza atendiendo a una serie de pasos básicos:

1. Escribir el argumento de la historia (Ejemplo practico: sección 2.3.1).
2. Extraer del argumento los diferentes componentes existentes en el framework: personajes jugadores, personajes no jugadores, habitaciones, y objetos (Ejemplo practico: sección 2.3.2).
3. Para cada componente especificar su comportamiento (Ejemplo practico: sección 2.3.2).
4. Describir el flujo de sucesos para la resolución exitosa de la misión (Ejemplo practico: sección 2.3.3).
5. Codificar cada objeto y personaje no jugador de forma independiente siguiendo la información dada en los dos anteriores puntos y usando las plantillas dadas en las secciones 2.2.2 y 2.2.3 (Ejemplo practico: sección 2.3.4).
6. Remodelación del mapa y ubicación de los nuevos objetos y personajes en él siguiendo los pasos descritos en 2.2.6 (Ejemplo practico: sección 2.3.5).
 - Los personajes jugadores, los personajes no jugadores y las puertas han de encontrarse en alguna habitación.
 - Los objetos pueden ubicarse tanto en habitaciones como en un inventario de un personaje jugador.
 - Las habitaciones vendrán identificadas por un ID único y deberán ser accesibles mediante una o más puertas.
7. Incluir los nuevos objetos necesarios de partida en los inventarios por defecto con los que se crea cada nuevo personaje jugador modificando `KernelBean.java` como se indica en la sección 2.2.7 (Ejemplo practico: sección 2.3.6).
8. Asignar una imagen que represente al objeto o personaje en cuestión como se explica en la sección 2.2.8 (Ejemplo practico: sección 2.3.7).

2.2.1. Componentes de una historia

Las historias se construyen básicamente a partir de objetos y personajes. Cada uno de estos viene implementado por distintas clases:

- Jugadores humanos. La clase que representa al jugador (el usuario) se denomina *ClienteJugador.java*. Esta clase no necesita ser modificada.
- Objetos pasivos del juego. Las puertas y demás objetos son del tipo *Plantilla_ObjetoNoPersonaje.java* (sección 2.2.2) y bastara con rescribir esta plantilla para implementarlos (el resultado ha de ubicarse en el paquete *agm.objetos.objetosNoPersonaje*).
- Actores del juego que no son jugadores humanos. El resto de personajes (los controlados por el servidor) son del tipo *Plantilla_PersonajeNoJugador.java* (sección 2.2.3) y habrá que reescribir esta plantilla para implementarlos (el resultado ha de ubicarse en el paquete *agm.objetos.personajesNoJugadores*).
- Las Habitaciones no vienen implementadas por clases en particular, si no que son creadas por la clase *GestorSistemaBean.java* (paquete *agm.servidor.kernel*) en su método *creaMundo()* al modelar el mundo distribuyéndolo en las distintas

habitaciones y ubicando los personajes y objetos en ellas, por lo tanto habrá que modificar esta clase para incluir o borrar las habitaciones (sección 2.2.6).

2.2.2. Crear nuevos objetos

Para la creación de los objetos haremos uso de la plantilla *Plantilla_ObjetoNoPersonaje.java* teniendo en cuenta las siguientes consideraciones:

- Por lo general un objeto tiene dos estados, pero esto es ampliable gracias a que es representado por un String.
- Las cuatro acciones que se pueden realizar con un objeto son “USAR”, “IR A”, “COGER” y “MIRAR”.
- Las acciones tienen genéricamente los siguientes argumentos:
 - o idA: Identificador de la acción genérica.
 - o idOd: Identificador del objeto sobre el que se está usando el objeto que implementa esta clase.
 - o idPJ: Identificador del jugador que ejecuta la acción.

Plantilla_ObjetoNoPersonaje.java :

```
package agm.objetos.objetosNoPersonaje;
import agm.servidor.habitaciones.*;
import agm.objetos.*;
import java.util.*;

public class Plantilla_ObjetoNoPersonaje extends ObjetoNoPersonajeClase{
    public void cambiaEstado(){
        //Aquí vendrá la lógica necesaria para cambiar de estado según el actual.
        //Por ejemplo:
        if (estado.equals("ESTADO-1")) {
            estado = "ESTADO-2";
        } else estado = "ESTADO-1";
    }
    public Acciones ejecutarAccion(String idA, String IdOd, String idPJ) {
        //Aquí vendrá la lógica necesaria para devolver la acción particular
        //correspondiente a la acción genérica que se quiere realizar.
        //Según cual sea la acción (y el estado actual si es necesario) devolvemos la acción implementada correspondiente
        if ("MIRAR".equals(idA)){
            if (this.estado.equals("ESTADO-1")){
                return new Acciones(){ //Generalmente la opción mirar devuelve una acción que muestra un texto descriptivo
                    //Función donde se define lo que sucederá al realizarse la ejecución
                    public void ejecutar(String idHab,String idPJ){
                        Habitacion h = conseguirHabitacion(idHab);
                        ObjetoActualizacion oA = generaNotificacion(idPJ,"TEXTO DESCRIPTIVO A MOSTRAR");
                        mostrarResultado(h,oA);
                    }
                    //Función que devuelve verdadero cuando la ejecución debe realizarse y falso cuando debe omitirse
                    public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ){return true;}
                };
            }
            else { ... }
        }
        else if ("COGER".equals(idA)){
            return new Acciones() {
                public void ejecutar (String idHab, String idPJ) {
                    Habitacion h = conseguirHabitacion(idHab);
                    if (interpretaCondicion(h,null,idPJ)){
                        //En el tercer argumento habrá que introducir la dirección del
                        //objeto que se quiere meter en el inventario, por ejemplo agm.objetos.objetosNoPersonaje.objeto
                        meterEnInventario(h,idPJ,"paquete.objeto","Mensaje para mostrar");
                    } else {
                        ObjetoActualizacion oa = generaNotificacion(idPJ,"No puedo hacer eso");
                        mostrarResultado(h,oa);
                    }
                }
            }
        }
    }
}
```

```

    }
    public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
        //Evaluar alguna condición en función de los parámetros
    }
};

}
else if ("USAR".equals(idA)){
    //Al realizar la acción USAR se hará sobre otro objeto (idO)
    //que habrá que buscar en la habitación (idHab)
    Habitacion h = Acciones.conseguirHabitacion(idHab);
    String nombre;
    ObjetoNoPersonajeClase oNPC = null;
    if (idO != null) {
        oNPC = h.buscarObjeto(idO, idPJ);
        if (oNPC != null) {
            String tipo = oNPC.getTipo();
            int ind = tipo.lastIndexOf(".");
            nombre = tipo.substring(ind + 1);
        }
        else nombre = "";
    }
    else nombre = "";
    //Según el objeto que sea se devolverá una acción u otra
    if (nombre.equals("Objeto_buscado")) {
        return new Acciones { ... }
    }
    else { ... }
}
}
else if ("IR A".equals(idA)){
    //Esta es la acción típica de las puertas o pasadizos
    Acciones a = new Acciones(){
        public void ejecutar(String idHab, String idPJ){
            Habitacion h = conseguirHabitacion(idHab);
            if (interpretaCondicion(h, null, idPJ)) {
                //En parametros.get(0) se encontrara el identificador de la habitación destino
                Habitacion hDest = conseguirHabitacion((String)parametros.get(0));
                cambioHabitacionCondicional(h, hDest, idPJ);
            }
            else {
                ObjetoActualizacion oA = generaNotificacion(idPJ, "No puedo ir alli");
                mostrarResultado(h, oA);
            }
        }
    };
    public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ){
        //Evaluar alguna condición en función de los parámetros
    }
};
LinkedList p=new LinkedList();
p.add(idHabDest);
a.setParametros(p);
return a;
}
}

public Plantilla_ObjetoNoPersonaje (String idO, String estado, String idHab, int posX, int posY) {
    super(idO, estado, idHab, posX, posY);
    tipo = this.getClass().getName();
}

public Plantilla_ObjetoNoPersonaje () {
    tipo = this.getClass().getName();
    estado="ESTADO-1";
}
}
}

```

2.2.3. Crear nuevos personajes

Los personajes son prácticamente como objetos cuya acción principal es HABLAR, la cual se caracteriza por la acción teRespondo que guía la conversación.

Para la implementación de los personajes se puede realizar modificando la plantilla *Plantilla_PersonajeNoJugador.java* :

```
package agm.objetos.personajesNoJugadores;
```

```

import agm.servidor.habitaciones.*;
import java.util.*;
import agm.objetos.objetosNoPersonaje.*;
import agm.objetos.*;

public class Plantilla_PersonajeNoJugador extends PersonajeNoJugadorClase{
    public Plantilla_PersonajeNoJugador(){tipo = this.getClass().getName();}
    public Plantilla_PersonajeNoJugador(String idPNJ,String nombre,String estado,String idHab,int posX,int posY) {
        super(idPNJ,nombre,idHab, posX,posY,estado);
        tipo = this.getClass().getName();
    }
    public void cambiaEstado(){
        if(estado.equals("CONVERSANDO")){
            estado="ESPERANDO";
        }
        else{
            estado = "CONVERSANDO";
        }
    }
    public Acciones ejecutarAccion(String idA, String idO, String idPj) {
        try{
            if ("HABLAR".equals(idA)){
                if(estado.equals("CONVERSANDO")){ //Si el personaje ya esta hablando con otro no debe empezar otra conversaci3n
                    return new Acciones(){
                        public void ejecutar(String idHab,String idPJ){
                            Habitacion h = conseguirHabitacion(idHab);
                            mostrarResultado(h,generaNotificacion(idPJ,"Est3 ocupado hablando con otro, tendr3 que esperar..."));
                        }
                        public boolean interpretaCondicion(Habitacion hab1,Habitacion hab2,String idPJ){return true;}
                    };
                }
                else{
                    Acciones a=teRespondo("OPCION INICIAL",idPj);
                    return a;
                }
            }
            else{
                return new Acciones(){
                    public void ejecutar(String idHab,String idPJ){
                        Habitacion h = conseguirHabitacion(idHab);
                        mostrarResultado(h,generaNotificacion(idPJ,"No"));
                    }
                    public boolean interpretaCondicion(Habitacion hab1,Habitacion hab2,String idPJ){return true;}
                };
            }
        }
        catch(Exception e){
            System.out.println("Excepci3n en ejecutarAccion");
            e.printStackTrace();
            return null;
        }
    }
    public Acciones teRespondo(String opcion,String idPJ){
        if (estado.equals("ESPERANDO")){
            estado="CONVERSANDO";
        }
        if (opcion.equals("OPCION INICIAL")){
            //estado="ESPERANDO"; //<- Se habr3 de descomentar esta l3nea si esta rama lleva al final de la conversaci3n
            return new Acciones(){
                public void ejecutar(String idHab,String idPJ){
                    Habitacion h = conseguirHabitacion(idHab);
                    ObjetoComunicacion o=new ObjetoComunicacion();
                    //Empieza la parte fija de la conversacion
                    LinkedList conversacion=new LinkedList();
                    //El argumento se pondra a true si empieza a hablar el usuario
                    o.setEmpiezaJugador(true);
                    conversacion.add("TEXTO QUE DICE EL JUGADOR");
                    conversacion.add("TEXTO QUE RESPONDE EL PNJ");
                    conversacion.add("TEXTO DEL JUGADOR");
                    conversacion.add("TEXTO QUE RESPONDE EL PNJ");
                    o.setConversacionFija(conversacion);
                    //Continua la parte en que se puede elejir que respuestas dar
                    LinkedList opciones=new LinkedList();
                    //Hay que colocar un identificador en la segunda opci3n para controlar el flujo de la conversaci3n
                    opciones.add(new Frase("RESPUESTAS POSIBLES", "ID DE LA OPCION"));
                }
            };
        }
    }
}

```

```

        o.setIdPJ(idPJ);
        //Damos el nombre del personaje para que salga en el dialogo antecediendo a sus frases
        o.setNombrePNJ("Plantilla_PersonajeNoJugador");
        mostrarResultado(h,o);
    }
    public boolean interpretaCondicion(Habitacion hab1,Habitacion hab2,String idPJ){return true;}
};
}
else if (opcion.equals("ID DE LA OPCION")){
    //estado="ESPERANDO"; //<- Se habrá de descomentar esta línea si esta rama lleva al final de la conversación
    return new Acciones(){
        public void ejecutar(String idHab,String idPJ){
            String idObj = null;
            Habitacion h = null;
            ObjetoComunicacion o = null;
            LinkedList conversacion = null;
            h = conseguirHabitacion(idHab);
            o=new ObjetoComunicacion();
            conversacion=new LinkedList();
            o.setEmpiezaJugador(TRUE ó FALSE);
            conversacion.add("TEXTO A MOSTRAR");
            o.setConversacionFija(conversacion);
            o.setIdPJ(idPJ);
            //Damos el nombre del personaje para que salga en el dialogo antecediendo a sus frases
            o.setNombrePNJ("Plantilla_PersonajeNoJugador");
            mostrarResultado(h,o);
        }
        public boolean interpretaCondicion(Habitacion hab1,Habitacion hab2,String idPJ)
        {return true;}
    };
}
else return null;
}
}
}

```

2.2.4. Acciones

Como se ha podido ver en los apartados anteriores, una parte fundamental tanto de los objetos como de los personajes no jugadores es las acciones que devuelven.

La acción viene definida por dos partes, la función *ejecutar* que es la que realiza la acción propiamente dicha y la función *interpretaCondicion* que es la que determina si se tiene que realizar la acción o no.

Los objetivos principales de las acciones son:

- **Mostrar un texto:** Este es el tipo de acción típica a devolver cuando se mira a un objeto o a un personaje. También se usa cuando se intenta por ejemplo hablar con un personaje ocupado para mostrar la negación.

```

return new Acciones(){
    public void ejecutar(String idHab,String idPJ){
        Habitacion h = conseguirHabitacion(idHab);
        mostrarResultado(h,generaNotificacion(idPJ,"TEXTO A MOSTRAR"));
    }
    public boolean interpretaCondicion(Habitacion hab1,Habitacion hab2,String idPJ){return true;}
};

```

- **Añadir un objeto al inventario:** Es la acción que ha de usarse cuando se quiere que un personaje obtenga un objeto. Normalmente responde a la acción “COGER” que se le aplica a un objeto, pero no es obligatorio que el objeto obtenido sea ese mismo, por ejemplo, si se aplica la acción coger a una taquilla puede obtenerse algo de su interior en vez de la taquilla en si.

```

return new Acciones() {
    public void ejecutar (String idHab, String idPJ) {
        Habitacion h = conseguirHabitacion(idHab);
        if (interpretaCondicion(h,null,idPJ)){
            //En el tercer argumento habrá que introducir la dirección del
            //objeto que se quiere meter en el inventario, por ejemplo agm.objetos.objetosNoPersonaje.objeto
            meterEnInventario(h,idPJ,"paquete.objeto","Mensaje para mostrar");
        } else {
            ObjetoActualizacion oa = generaNotificacion(idPJ,"No puedo hacer eso");
            mostrarResultado(h,oa);
        }
    }
    public boolean interpretaCondicion(Habitacion hab1,Habitacion hab2,String idPJ){
        //Evaluar alguna condición en función de los parámetros
    }
}

```

- **Mover al personaje de habitación:** Es la acción típica de las puertas pero puede elegirse que sea desencadenada por otro motivo, por ejemplo para simular un conjuro de teletransportación.

```

Acciones a = new Acciones(){
    public void ejecutar(String idHab,String idPJ){
        Habitacion h = conseguirHabitacion(idHab);
        if (interpretaCondicion(h,null,idPJ)) {
            //En parametros.get(0) se encontrara el identificador de la habitación destino
            Habitacion hDest = conseguirHabitacion((String)parametros.get(0));
            cambioHabitacionCondional(h,hDest,idPJ);
        } else {
            ObjetoActualizacion oA = generaNotificacion(idPJ,"No puedo ir alli");
            mostrarResultado(h,oA);
        }
    }
    public boolean interpretaCondicion(Habitacion hab1,Habitacion hab2,String idPJ){
        //Evaluar alguna condición en función de los parámetros
    }
};
LinkedList p=new LinkedList();
p.add(idHabDest);
a.setParametros(p);

```

2.2.5. Aumentar la funcionalidad de objetos y personajes.

Lo visto hasta ahora seria una implementación básica de un objeto o personaje, sin embargo se le puede añadir otras funcionalidades como que el objeto (o personaje no jugador) mantenga una lista de los personajes jugadores que han pasado por él para así no permitir al mismo realizar la misma acción varias veces por ejemplo.

También se puede incluir la interacción con otros elementos del juego como otros personajes no jugadores almacenando su identificador para luego buscar dicho personaje y en función de él actuar de una manera u otra.

Para ambas opciones hay que incluir rutinas que utilizan xml para mantener la persistencia de estos nuevos datos.

El resultado seria incluir las siguientes líneas de código:

- Librerías necesarias

```

import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.dom.DOMSource;
import org.w3c.dom.*;
import java.io.File;

```

- Atributos nuevos

```
private String idPersonaje; //Identificar del personaje con el que se quiere interactuar
private LinkedList PJs; //Lista de los jugador que han pasado por este objeto
```

Ahora teniendo el identificador del personaje con el que queremos interactuar podremos incluir en las condiciones de las acciones que se busque a este personaje y se consulte sobre él alguna propiedad para decidir que hacer, como podemos ver en las líneas en negrita de la función *interpretaCondicion* del siguiente ejemplo.

Además podemos ver como se utiliza la lista de personajes para saber si alguien ha cogido ya el objeto y en tal caso no dárselo otra vez (líneas en negrita de la primera parte del if).

```
if ("COGER".equals(idA)){
    if (!(PJs.contains(IdPJ))){
        return new Acciones() {
            public void ejecutar (String idHab, String idPJ) {
                Habitacion h = conseguirHabitacion(idHab);
                if (interpretaCondicion(h,null,idPJ)){
                    PJs.add(idPJ);
                    guardarLista();
                    meterEnInventario(h,idPJ,"agm.objetos.objetosNoPersonaje.Carnel","Ya tengo mi falsa identidad");
                } else {
                    ObjetoActualizacion oa = generaNotificacion(idPJ,"Deberia esperar a que se ausentara el secretario");
                    mostrarResultado(h,oa);
                }
            }
        };
    }
    public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2,String idPJ) {
        try {
            agm.objetos.personajesNoJugadores.PersonajeNoJugadorClase pnj;
            pnj = hab1.buscarPNJEnHabitacion(idSecretario);
            if ((pnj.getEstado()).equals("AUSENTE")) {
                pnj.setEstado("ESPERANDO");
                return true;
            } else { return false;}
        } catch (Exception e) {
            return false;
        }
    }
}
```

- Persistencia xml

Para mantener los datos entre conexión y conexión será necesario la utilización de persistencia xml la cual utilizaremos con las siguientes funciones nuevas.

```
public void cargarLista(){
    PJs=new LinkedList();
    Document document = null;
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    try {
        DocumentBuilder builder = factory.newDocumentBuilder();
        document = builder.parse( new File(getIdO()+".xml"));
        PJs=new LinkedList();
        Node raiz=document.getFirstChild().getNextSibling();
        if(raiz.getFirstChild()!=null){
            Node personaje=raiz.getFirstChild().getNextSibling();
            while(personaje!=null){
                String idP=personaje.getFirstChild().getNodeValue();
                personaje=personaje.getNextSibling().getNextSibling();
                PJs.add(idP);
            }
        }
    }
```

```

    }
}
catch(Exception e){
    System.out.println("Error al cargar la lista de personajes");
    e.printStackTrace();
    PJs=new LinkedList();
}
}
public void guardarLista(){
    try{
        DocumentBuilder builder=DocumentBuilderFactory.newInstance().newDocumentBuilder();
        Document doc=builder.newDocument();
        Element personajes=doc.createElement("Personajes");
        doc.appendChild(personajes);
        int longitud=PJs.size();
        for(int i=0;i<longitud;i++){
            Element personaje=doc.createElement("Personaje");
            personajes.appendChild(personaje);
            personaje.appendChild(doc.createTextNode((String)PJs.get(i)));
        }
        Transformer trans=TransformerFactory.newInstance().newTransformer();
        trans.setOutputProperty(OutputKeys.ENCODING,"ISO-8859-1");
        trans.setOutputProperty(OutputKeys.INDENT,"yes");
        trans.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM,"personajes.dtd");
        trans.transform(new DOMSource(doc),new StreamResult(new File(getIdO()+".xml")));
    }
    catch(Exception e){
        System.out.println("Error al guardar la lista de personajes");
        e.printStackTrace();
    }
}
public void setIdO(String string) {
    super.setIdO(string);
    cargarLista();
}
public void setPJs(LinkedList lista) {
    PJs=lista;
    guardarLista();
}
public void cargar_idPersonaje(){
    Document document = null;
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    try {
        DocumentBuilder builder = factory.newDocumentBuilder();
        document = builder.parse( new File(getIdO()+".xml"));
        Node raiz=document.getFirstChild().getNextSibling();
        if(raiz!=null){
            idPersonaje=raiz.getFirstChild().getNodeValue();
        }
    }
    catch(Exception e){
        System.out.println("Error al cargar el idPersonaje");
        e.printStackTrace();
        idPersonaje="";
    }
}
public void guardar_idPersonaje(){
    try{
        DocumentBuilder builder=DocumentBuilderFactory.newInstance().newDocumentBuilder();
        Document doc=builder.newDocument();
        //Hay que pasarle como argumento el nombre de la clase que implementa el personaje
        //cuyo id queremos almacenar
        Element e_idPersonaje=doc.createElement("Nombre_de_la_clase_del_idPersonaje");
        doc.appendChild(e_idPersonaje);
        e_idPersonaje.appendChild(doc.createTextNode(idPersonaje));
        Transformer trans=TransformerFactory.newInstance().newTransformer();
        trans.setOutputProperty(OutputKeys.ENCODING,"ISO-8859-1");
        trans.setOutputProperty(OutputKeys.INDENT,"yes");
        //Hay que pasarle como argumento un dtd que tenga como nombre el de
        //la clase que implementa el personaje cuyo id queremos almacenar
        trans.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM,"Nombre_de_la_clase_del_idPersonaje.dtd");
        trans.transform(new DOMSource(doc),new StreamResult(new File(getIdO()+".xml")));
    }
    catch(Exception e){
        System.out.println("Error al guardar el idPersonaje");
    }
}

```



```
e.printStackTrace();
}
}
```

El .dtd al que se hace mención deberá estar colocado en el directorio bin del jboss con la siguiente sintaxis:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT Nombre_de_la_clase_del_idPersonaje (#PCDATA)>
```

2.2.6. Modificar el mapa

Inicialmente se parte del mapa ejemplificado por este esquema:

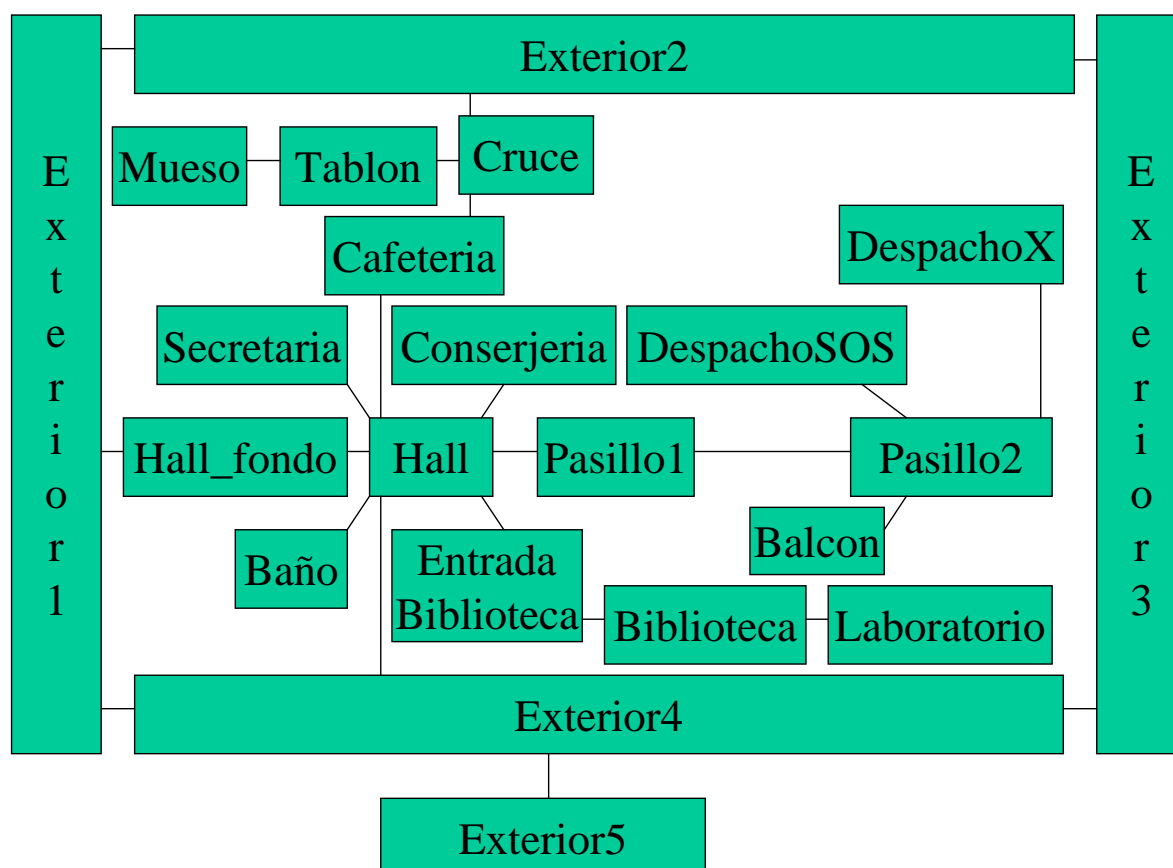


Ilustración 1. Mapa de habitaciones por nombre.

Cada habitación tiene un identificador:

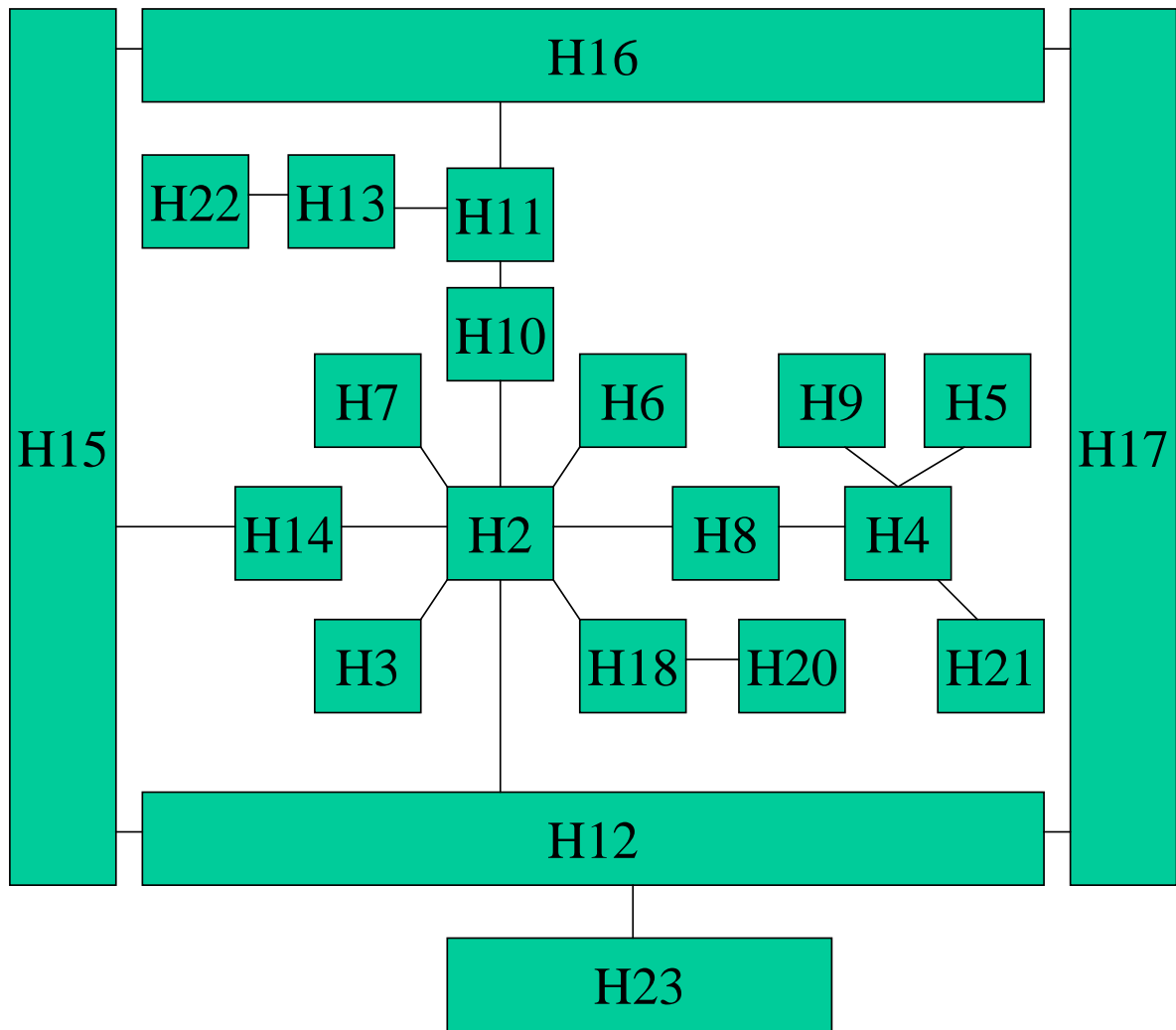


Ilustración 2. Mapa de habitaciones por identificadores

A parte de estas habitaciones hay una incomunicada denominada *Casa* y que tiene por identificador el valor *H1*, se trata de una habitación para la gestión interna del juego y que no hay que manipular.

Los identificadores de las habitaciones han de ser únicos ya que se trataran las habitaciones en base a este parámetro.

La manera de crear una habitación nueva es incluyendo una nueva función en *GestorSistemaBean.java* de este tipo (los string y enteros con los que se inicializan las variables son a modo de ejemplo, habrá que sobrescribirlo según cada caso):

```
//Redefínase el nombre del método de manera mnemotécnica
private void crearHabitacionNueva(){
    String idHab = "HX"; //Identificador de la habitación que vamos a crear
    String nombre = "HabitacionNueva"; //Nombre de la habitación
    boolean luz = true; //Indicamos si la habitación tiene la luz encendida o no
    int ancho = 12; // Ancho de la habitación
    int alto = 7; //Alto de la habitación
    String mascara = "0.238-970.523"; //Mascara de la imagen de fondo
    LinkedList objetos = new LinkedList(); //Lista con los objetos que contiene la habitación
    LinkedList pnjs = new LinkedList(); //Lista con los personajes que contiene la habitación
    LinkedList puertas = new LinkedList(); //Lista con las puertas a otras habitaciones
    //Si queremos que la habitación no quede aislada habrá que ponerle al menos una puerta
    String idHabDest = "HY"; //identificador de la habitación destino
    int posX = 700; //Coordenada X de la puerta
```

```

int posY = 400; //Coordenada Y de la puerta
String estado = "ABIERTA"; // Este valor ha de ser "ABIERTA" o "CERRADA" y indica obviamente el estado de la puerta
String idO="O"+ contObjetos++; //Como objeto que es la puerta hay que darle un identificador, por nomenclatura elegimos la
letra O seguida de el correspondiente numero secuencial
puertas.add (new Puerta(idO, idHab, idHabDest, posX, posY, estado)); // Finalmente se añade la puerta a la lista
//Finalmente podemos agregar los demás objetos de la habitación
//Como cada objeto tiene su propia clase y por lo tanto su propio constructor, esta parte puede variar en función de los parámetros
que tenga e inicialización que sea requerida, pero por lo general esta es la parte invariante:
posX = 20; // Coordenada X del objeto
posY = 10; //Coordenada Y del objeto
estado = "ESTADO"; //Estado del objeto (dependerá de cada objeto)
idO="O"+ contObjetos++; //Identificador del objeto
objetos.add (new ObjetoEnCuestion (idO, estado, idHab, posX, posY)); //Añadimos el objeto a la lista de objetos de la habitación
crearHabitacion (idHab, nombre, luz, ancho, alto, mascara, puertas, objetos, pnjs); //Finalmente creamos la habitación pasándole
todos los parámetros inicializados anteriormente
}

```

Por último insertaremos la llamada de la función recién creada al final de la función *creaMundo* de *GestorSistemaBean*.

Hay que tener en cuenta que la *Puerta* que se utiliza en el anterior extracto de código es el tipo de puerta genérica de funcionalidad mínima (comunicar dos habitaciones), si se quisiera una mas sofisticada (como es el caso de la puerta a conserjería que solo se puede traspasar si el conserje esta ocupado) habría que extender la clase *Puerta* e implementarla específicamente.

2.2.7. Añadir objetos iniciales en el inventario de un jugador

Si se desea que los jugadores recién creados partan con un inventario predefinido basta con incluir en la clase *KernelBean.java* del paquete *agm.servidor.kernel* modificaciones en la función *objetosIniciales()*.

De esta manera se consigue que el objeto *Llave_taquilla* se encuentre inicialmente en el personaje al crearlo:

```

public LinkedList objetosIniciales() {
    LinkedList ob = new LinkedList();
    Llave_taquilla lt = new Llave_taquilla();
    lt.setIdO(KernelUtil.generateGUID(lt));
    ob.add(lt);
    //Incluir aquí mas objetos que se desean como iniciales.
    return ob;
}

```

2.2.8. Representación grafica

La representación grafica de los objetos se hace por la carga dinámica de las imágenes que se encuentran en el directorio `\src\agm\clienteJugador\interfazGrafica\imagenes` bajo las siguientes reglas:

- A un objeto (tanto personajes como no personajes) de nombre *NOMBRE* le corresponde en el estado *ESTADO* la imagen *NOMBRE_ESTADO.gif*
- A una habitación cuyo identificador es *HX* le corresponde como fondo la imagen *HX.jpg*

2.3. Aplicación a un caso práctico: 10% de Azar

En esta sección se describe una historia y su proceso de implementación siguiendo el framework AGM.

2.3.1. Descripción global de la historia

La historia se llama *10% de Azar*. En esta historia el jugador es un alumno que descubre, al hablar con su profesor, que ha suspendido la asignatura SOS no por sus malos resultados académicos si no porque el profesor solo ha aprobado al mínimo indispensable, es decir, al 10%.

El objetivo del alumno será por lo tanto conseguir entrar en ese 10% para aprobar. Para ello se servirá del carné de otro estudiante (uno que si a aprobado) aprovechando que el profesor no reconoce físicamente a los alumnos y le caen especialmente mal los alumnos que aun aprobando van a la revisión, hasta el punto de cambiarles la nota a un suspenso. La idea es conseguir que suspenda al otro alumno para que el jugador pase a ser aprobado automáticamente en su lugar ya que ha de seguir manteniéndose la estadística de que haya un 10% de aprobados.

El alumno podrá buscar el carné usurpado que necesita entre los carnés perdidos que se guardan en secretaria pero para ello necesitara deshacerse del secretario garantizando su impunidad.

2.3.2. Definición de personajes y objetos

- Personajes jugador:
El personaje jugador es el alumno de la asignatura SOS impartida por el profesor B. Nos referiremos a este alumno como J.
- Personajes no jugador:

Nombre: Profesor SOS	ID: B
Descripción: Es el profesor de la asignatura SOS al que tenemos que engañar para conseguir el aprobado.	
Estados: <ul style="list-style-type: none">➔ Inicial: ESPERANDO➔ Secundario: CONVERSANDO Acciones: <ul style="list-style-type: none">➔ MIRAR: Muestra “Es maligno... se atreve a suspenderme cuando él ni siquiera sabe diferenciar a los alumnos”➔ HABLAR:<ul style="list-style-type: none">○ Si ESPERANDO: Inicia la conversación del punto 1.1 del flujo de sucesos (sección 2.3.3).○ Si CONVERSANDO: Muestra “Ya sea por su paciencia o por su capacidad mental, este hombre no da para hablar con dos personas a la vez, mejor esperar”	
Ubicación: Habitación “DespachoSOS” (H9).	

Clase que lo implementa: ProfesorSOS.java	Paquete: agm.objetos.personajesNoJugadores
Plantilla: Plantilla_PersonajeNoJugador.java	

Nombre: Secretario	ID: C
Descripción: Es el empleado de secretaria del que habrá que zafarse para poder robar en la secretaria el carné que necesitamos.	
Estados: → Inicial: ESPERANDO → Secundario: CONVERSANDO → Secundario: AUSENTE Acciones: → MIRAR: Muestra “Cuenta la leyenda que esta para ayudar, pero lo único cierto es que aumentan la clientela de aspirinas bayer” → HABLAR: <ul style="list-style-type: none"> ○ Si ESPERANDO: Inicia la conversación del punto 1.9.2.1 del flujo de sucesos (sección 2.3.3). ○ Si CONVERSANDO: Muestra “Vaya, ahora esta ocupado, esperare un poco, no creo que el que esta solicitando ayuda tarde mucho en desesperar” ○ Si AUSENTE: Muestra “Debí desconfiar, esta ausente, lo que veo solo es la estela que dejo en su huida” 	
Ubicación: Habitación “Secretaria” (H7).	
Clase que lo implementa: Secretario.java	Paquete: agm.objetos.personajesNoJugadores
Plantilla: Plantilla_PersonajeNoJugador.java	

- Objetos:

Nombre: Taquilla	ID: TJ
Descripción: Taquilla del personaje donde se encuentran sus efectos personales como por ejemplo el carné de estudiante.	
Estados: → Inicial: CERRADA → Secundario: ABIERTA Acciones: → MIRAR: muestra “Esta es mi taquilla, donde guardo mis cosas” → USAR con LLTJ: Conmuta el estado de TJ. → COGER: <ul style="list-style-type: none"> ○ Si CERRADA: muestra “Para coger mis cosas debería primero abrir la taquilla” ○ Si ABIERTA: añade al inventario CEIJ. 	
Ubicación: Pasillo1 (H8)	
Clase que lo implementa: Taquilla.java	Paquete: agm.objetos.objetosNoPersonaje
Plantilla: Plantilla_ObjetoNoPersonaje.java	

Nombre: Llave de la taquilla	ID: LLTJ
Descripción: Con esta llave podremos abrir o cerrar nuestra taquilla.	

Estados: → Inicial: ESTADOUNICO	
Acciones: → MIRAR: Muestra “Esta es la llave que abre mi taquilla del pasillo”. → USAR con TJ: Conmuta el estado de TJ.	
Ubicación: Inventario del jugador.	
Clase que lo implementa: Llave_taquilla.java	Paquete: agm.objetos.objetosNoPersonaje
Plantilla: Plantilla_ObjetoNoPersonaje.java	

Nombre: CarneJ	ID: CEJ
Descripción: Carné del estudiante J que le sirve para identificarse en el ámbito de la facultad.	
Estados: → Inicial: ESTADOUNICO	
Acciones: → MIRAR: muestra “Este es mi carné de estudiante” → USAR con B: Inicia la conversación detallada en el punto 1.7 del flujo de sucesos (sección 2.3.3).	
Ubicación: Taquilla (TJ).	
Clase que lo implementa: CarneJ.java	Paquete: agm.objetos.objetosNoPersonaje
Plantilla: Plantilla_ObjetoNoPersonaje.java	

Nombre: Caja de carnés I	ID: CCI
Descripción: En esta caja encontraremos los carnes I que nos servirán para suplantar a dicho alumno.	
Estados: → Inicial: ESTADOUNICO	
Acciones: → MIRAR: muestra “Aquí es donde se guardan los carnés perdidos” → COGER: Mete en el inventario del jugador un CEI.	
Ubicación: Habitación Secretaria (H7).	
Clase que lo implementa: CajaDeCarnes.java	Paquete: agm.objetos.objetosNoPersonaje
Plantilla: Plantilla_ObjetoNoPersonaje.java	

Nombre: CarneI	ID: CEI
Descripción: Carné del estudiante I.	
Estados: → Inicial: ESTADOUNICO	
Acciones: → MIRAR: muestra “Este es el carné de estudiante de I, todo un héroe sin saberlo” → USAR con B: Inicia la conversación detallada en el punto 1.7 del flujo de sucesos (sección 2.3.3).	
Ubicación: Se encuentra en CCI ya que se obtiene al realizar la acción coger sobre tal objeto pero realmente no esta situado en ninguna parte a la hora de implementar.	
Clase que lo implementa: CarneI.java	Paquete: agm.objetos.objetosNoPersonaje
Plantilla: Plantilla_ObjetoNoPersonaje.java	

Nombre: AprobadoSOS	ID: ASOS
Descripción: Carné del estudiante I.	
Estados: → Inicial: ESTADOUNICO Acciones: → MIRAR: muestra “Es el aprobado SOS”	
Ubicación: Solo aparece tras complementar la misión y se queda en su inventario.	
Clase que lo implementa: AprobadoSOS.java	Paquete: agm.objetos.objetosNoPersonaje
Plantilla: Plantilla_ObjetoNoPersonaje.java	

NOTA: Las acciones que no han sido detalladas desencadenan el un mensaje de negación (por ejemplo: “No puedo hacer eso”) ya que son acciones que no tienen sentido en el contexto de la historia.

2.3.3. Flujo de sucesos

1. J va al despacho (H9) del profesor e inicia una conversación.
 - 1.1. “Hola, venia por la revisión de SOS...”
 - 1.2. “¿Tu otra vez?”
 - 1.3. “¿Qué?”
 - 1.4. “Uffff... Todos los alumnos sois iguales... a ver, muéstrame tu carné de estudiante a ver si te tengo ya apuntado... como seas uno de esos alumnos que vienen a incordiar mas de una vez intentando rascar alguna décima te bajare la nota aun más de cómo la tengas”
 - 1.5. Si el alumno J tiene su carné de estudiante (CEJ) podrá elegir mostrárselo y proseguir la conversación como indica el punto 1.6. En caso contrario deberá proceder de la siguiente manera:
 - 1.5.1. J sale del despacho (H9) para ir al pasillo2 (H4) y de ahí pasar al pasillo1 (H8).
 - 1.5.2. Abre su taquilla (TJ) usando su llave (LLTJ) sobre ella y coge su carné de estudiante (CEJ), el cual pasa a figurar en su inventario.
 - 1.5.3. Ahora puede volver al despacho retomando el punto 1.
 - 1.6. J muestra a B el CEJ
 - 1.7. Opciones:
 - 1.7.1. Si es la primera vez:
 - 1.7.1.1. “Vale... veamos tu examen... uuuuuufff... un completo desastre, vamos, como todos los demás exámenes, si fuera por mí estabais todos suspensos, pero como tengo que aprobar al menos a un 10%...”
 - 1.7.1.2. “Pero eso es injusto, si usted hubiese puesto un examen medianamente parecido a lo que dio en clase esto no habría pasado”
 - 1.7.1.3. “La vida no es justa chaval, pero no te equivoques, no vas a suspender porque yo sea un mal profesor, sino que vas a suspender porque eres el onceavo alumno, jajajaja”
 - 1.7.1.4. “¿Qué?”
 - 1.7.1.5. “Si, sois 100 en clase y he aprobado a los 10 primeros por orden alfabético, jaja, si simplemente tu apellido empezara por I en vez de por J estarías aprobado... ¡Que demonios! Si el alumno I cuando

- estuvo aquí implorando que no le jodiera la media de la carrera no hubiera desistido a tiempo ahora tu estarías aprobado”
- 1.7.1.6. “¡¡Esto no quedara así!!”
 - 1.7.2. Si no es la primera vez:
 - 1.7.2.1. “Arrrrrrff... te lo dije, no hay nada que me moleste mas que un alumno pesado. Enhorabuena le acabas de regalar tu nota al siguiente en la lista, la tuya será ahora un 0”
 - 1.8. J sale del despacho, pasa pro el pasillo1 (H8), el pasillo2 (H4), va al HALL (H2) y de hay a Secretaria (H7)
 - 1.9. Opciones:
 - 1.9.1. Si el empleado de secretaria (C) esta atendiendo a alguien o simplemente se encuentra allí en espera al intentar coger el CEI saltara el mensaje “Imposible con el de secretaria delante, antes tengo que conseguir que se vaya”
 - 1.9.2. Si el empleado de secretaria (C) esta en espera J deberá empezar una conversación con él.
 - 1.9.2.1. “Hola”
 - 1.9.2.2. “¿Qué quieres?”
 - 1.9.2.3. Opciones.
 - 1.9.2.3.1. “Le he estado observando, es increíble la labor que esta llevando aquí, debe estar exhausto”
 - 1.9.2.3.1.1. “Si, es verdad. Será mejor que vaya a la cafetería a descansar un poco, que si no luego no rindo y muchos alumnos se verían perjudicados”
 - 1.9.2.3.1.2. Saltamos al punto 1.9.3.
 - 1.9.2.3.2. “Necesito información sobre las becas”
 - 1.9.2.3.2.1. Saltamos al punto 1.9.2.4.
 - 1.9.2.3.3. “Necesito información sobre la matricula”
 - 1.9.2.3.3.1. Saltamos al punto 1.9.2.4.
 - 1.9.2.3.4. “Vengo de Yugoslavia, he llegado hoy mismo, me he perdido y he aparecido aquí, ¿podría ayudarme a llegar a la embajada?”
 - 1.9.2.3.4.1. Saltamos al punto 1.9.2.4.
 - 1.9.2.4. “Eso esta en la web”
 - 1.9.2.5. “No, ya he mirado ahí”
 - 1.9.2.6. “Te digo que esta en la web”
 - 1.9.2.7. “Lo he mirado 200 veces y no esta”
 - 1.9.2.8. “Pues esta en los tablones”
 - 1.9.2.9. “No, tampoco esta ahí”
 - 1.9.2.10. “Tiene que estar”
 - 1.9.2.11. “Pues salga a buscarlo a ver si lo encuentra, porque yo llevo dos horas y no he visto nada que me sirva de ayuda”
 - 1.9.2.12. “Uuuuum... espera aquí un momento, voy a preguntarle a mi compañero a ver si sabe algo”
 - 1.9.2.13. Pasamos al punto 2.9.3.
 - 1.9.3. Si no esta el empleado de secretaria (C) cogemos el carné del alumno I (CEI) y huimos con el objeto en nuestro inventario.
 2. Ahora que tenemos el CEI volvemos al punto 2. y cuando llegamos al punto 2.6. mostramos el CEI en vez del CEJ consiguiendo así que suspenda I y provocando nuestro aprobado.
 3. Se mete en el inventario el aprobado de la asignatura SOS.

4. ¡¡Misión superada!!

Flujo de sucesos resumido por acciones:

1. J IR A H9
 - 1.1. J HABLAR B
 - 1.5.1. J IR A H4, J IR A H8
 - 1.5.2. J USAR LLTJ con TJ
 - 1.6. J USAR CEI/CEJ con B
 - 1.7. Elegir opción
 - 1.8. J IR A H2 y J IR A H7
 - 1.9.2. J HABLAR C
 - 1.9.2.3. Elegir opción
 - 1.9.3. J COGER CEI
2. J IR A H2, J IR A H8, J IR A H4, J IR H9

2.3.4. Codificación de objetos y personajes

La codificación de los objetos y personajes ha de hacerse siguiendo los detalles descritos en la 2.3.2 y 2.3.3.

A continuación se expondrá el código particular de cada clase con negrita en las líneas que fuero modificadas sobre la plantilla.

Profesor SOS (B)

Código correspondiente (ProfesorSOS.java):

```
package agm.objetos.personajesNoJugadores;

import java.io.File;
import java.util.LinkedList;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import agm.objetos.NotificacionTextual;
import agm.objetos.ObjetoActualizacion;
import agm.objetos.ObjetoComunicacion;
import agm.objetos.objetosNoPersonaje.Acciones;
import agm.objetos.objetosNoPersonaje.ObjetoNoPersonajeClase;
import agm.servidor.habitaciones.Habitacion;

public class ProfesorSOS extends PersonajeNoJugadorClase {
    private LinkedList PJs;

    public ProfesorSOS() {
        tipo = this.getClass().getName();
        PJs = new LinkedList();
        PJs.add("Alumno_I");
    }

    public ProfesorSOS(String idPNJ, String nombre, String estado,
        String idHab, int posX, int posY) {
```

```

super(idPNJ, nombre, idHab, posX, posY, estado);
tipo = this.getClass().getName();
PJs = new LinkedList();
PJs.add("Alumno_I");
}

public void cambiaEstado() {
    if (estado.equals("CONVERSANDO")) {
        estado = "ESPERANDO";
    } else {
        estado = "CONVERSANDO";
    }
}

public Acciones ejecutarAccion(String idA, String idO, String idPj) {
    try {
        if ("HABLAR".equals(idA)) {
            if (estado.equals("CONVERSANDO")) { //esta hablando con otro
                return new Acciones() {
                    public void ejecutar(String idHab, String idPJ) {
                        Habitacion h = conseguirHabitacion(idHab);
                        mostrarResultado(h,
                            generaNotificacion(idPJ,
                                "Ya sea por su paciencia o por su capacidad mental, \n"
                                + "este hombre no da para hablar con dos personas a la vez, mejor esperar"));
                    }
                    public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
                        return true; }
                };
            } else { //esta libre
                Acciones a = teRespondo("OPCION INICIAL", idPj);
                return a;
            }
        } else if ("MIRAR".equals(idA)) {
            return new Acciones() {
                public void ejecutar(String idHab, String idPJ) {
                    Habitacion h = conseguirHabitacion(idHab);
                    NotificacionTextual n = new NotificacionTextual("Es maligno... se atreve a suspenderme cuando él ni
siquiera sabe diferenciar a los alumnos", idPJ);
                    ObjetoActualizacion oA = new ObjetoActualizacion();
                    LinkedList lista = new LinkedList();
                    lista.add(n);
                    oA.setNotificacionesTextuales(lista);
                    mostrarResultado(h, oA);
                }
                public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
                    return true;
                }
            };
        } else if ("USAR".equals(idA)) {
            Habitacion h = conseguirHabitacion(idHab);
            String nombre;
            ObjetoNoPersonajeClase oNPC = null;
            if (idO != null) {
                oNPC = h.buscarObjeto(idO, idPj);
                if (oNPC != null) {
                    String tipo = oNPC.getTipo();
                    int ind = tipo.lastIndexOf(".");
                    nombre = tipo.substring(ind + 1);
                } else nombre = "";
            } else nombre = "";
            Acciones a = null;
            if ((nombre.equals("CarneJ")) && (estado.equals("ESPERANDO"))) {
                a = teRespondo("OPCION 2.7", idPj);
            }
            if ((nombre.equals("CarneI")) && (estado.equals("ESPERANDO"))) {
                PJs.add(idPj);
                guardarLista();
                a = teRespondo("OPCION 2.7b", idPj);
            }
            return a;
        } else {
            return new Acciones() {
                public void ejecutar(String idHab, String idPJ) {
                    Habitacion h = conseguirHabitacion(idHab);

```

```

        mostrarResultado(h, generaNotificacion(idPJ, "No puedo hacer eso"));
    }
    public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
        return true;
    }
};

}
} catch (Exception e) {
    System.out.println("Excepcion en ejecutarAccion");
    e.printStackTrace();
    return null;
}
}

public Acciones teRespondo(String opcion, String idPJ) {
    if (estado.equals("ESPERANDO")) estado = "CONVERSANDO";
    if (opcion.equals("OPCION INICIAL")) {
        estado = "ESPERANDO"; // Siempre se cambia el estado si la conversacion acaba por esta rama.
        return new Acciones() {
            public void ejecutar(String idHab, String idPJ) {
                Habitacion h = conseguirHabitacion(idHab);
                ObjetoComunicacion o = new ObjetoComunicacion();
                LinkedList conversacion = new LinkedList();
                o.setEmpiezaJugador(true);
                conversacion.add("Hola, venia por la revision de SOS...");
                conversacion.add("¿Tu otra vez?");
                conversacion.add("¿Que?");
                conversacion.add("Uffff... Todos los alumnos sois iguales... a ver,\n"
                    + "muestrame tu carne de estudiante a ver si te tengo ya apuntado...\n"
                    + "como seas uno de esos alumnos que vienen a incordiar mas de una vez\n"
                    + "intentando rascar alguna decima te bajare la nota aun mas de como la tengas");
                o.setConversacionFija(conversacion);
                o.setIdPJ(idPJ);
                o.setNombrePNJ("ProfesorSOS");
                mostrarResultado(h, o);
            }
            public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
                return true;
            }
        };
    }
    if (opcion.equals("OPCION 2.7") || opcion.equals("OPCION 2.7b")) {
        if (PJs.contains(idPJ)) { //ya tiene el aprobado, conversacion 2
            estado = "ESPERANDO";
            if (opcion.equals("OPCION 2.7b")) {
                return new Acciones() {
                    public void ejecutar(String idHab, String idPJ) {
                        Habitacion h = conseguirHabitacion(idHab);
                        ObjetoComunicacion o = new ObjetoComunicacion();
                        LinkedList conversacion = new LinkedList();
                        o.setEmpiezaJugador(false);
                        conversacion.add("Arrrrrff... te lo dije, no hay nada que me moleste mas que un alumno pesado.
\nEnhorabuena le acabas de regalar tu nota al siguiente en la lista, la tuya sera ahora un 0");
                        o.setConversacionFija(conversacion);
                        o.setIdPJ(idPJ);
                        o.setNombrePNJ("ProfesorSOS");
                        mostrarResultado(h, o);
                        meterEnInventario(h, idPJ, "agm.objetos.objetosNoPersonaje.AprobadoSOS", "¡Aprobe!");
                    }
                    public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
                        return true;
                    }
                };
            }
        } else {
            return new Acciones() {
                public void ejecutar(String idHab, String idPJ) {
                    Habitacion h = conseguirHabitacion(idHab);
                    ObjetoComunicacion o = new ObjetoComunicacion();
                    LinkedList conversacion = new LinkedList();
                    o.setEmpiezaJugador(false);
                    conversacion.add("Arrrrrff... te lo dije, no hay nada que me moleste mas que un alumno
pesado.\nEnhorabuena le acabas de regalar tu nota al siguiente en la lista, la tuya sera ahora un 0");
                    o.setConversacionFija(conversacion);
                    o.setIdPJ(idPJ);
                    o.setNombrePNJ("ProfesorSOS");
                }
            };
        }
    }
}

```

```

        mostrarResultado(h, o);
    }
    public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
        return true;
    }
}
};

} else { //No ha venido antes
    estado = "ESPERANDO";
    return new Acciones() {
        public void ejecutar(String idHab, String idPJ) {
            Habitacion h = conseguirHabitacion(idHab);
            ObjetoComunicacion o = new ObjetoComunicacion();
            LinkedList conversacion = new LinkedList();
            o.setEmpiezaJugador(false);
            conversacion .add("Vale... veamos tu examen... uuuuuufff... un completo desastre, vamos, como todos los
demás exámenes,\n"
+ "si fuera por mi estabais todos suspensos, pero como tengo que aprobar al menos a un 10%...");
            conversacion .add("Pero eso es injusto, si usted hubiese puesto un examen medianamente parecido\n"
+ "a lo que dio en clase esto no habra pasado");
            conversacion .add("La vida no es justa chaval, pero no te equivoques, no vas a suspender porque yo sea un
mal profesor,\n"
+ "sino que vas a suspender porque eres el onceavo alumno, jajajaja");
            conversacion.add("¿Que?");
            conversacion .add("Si, sois 100 en clase y he aprobado a los 10 primeros por orden alfabetico, jaja,\n"
+ "si simplemente tu apellido empezara por I en vez de por J estarias aprobado...\n"
+ "¿Que demonios! Si el alumno I cuando estuvo aqui implorando que no le jodiera la media de\n"
+ "la carrera no hubiera desistido a tiempo ahora tu estarias aprobado");
            conversacion.add("¡¡Esto no quedara asi!!");
            o.setConversacionFija(conversacion);
            o.setIdPJ(idPJ);
            o.setNombrePNJ(nombre);
            mostrarResultado(h, o);
            PJs.add(idPJ);
        }
        public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
            return true;
        }
    };
}
} else return null;
}

public void cargarLista() {
    PJs = new LinkedList();
    Document document = null;
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    try {
        DocumentBuilder builder = factory.newDocumentBuilder();
        document = builder.parse(new File(getIdPNJ() + ".xml"));
        PJs = new LinkedList();
        Node raiz = document.getFirstChild().getNextSibling(); //personajes
        if (raiz.getFirstChild() != null) {
            Node personaje = raiz.getFirstChild().getNextSibling(); //personaje
            while (personaje != null) {
                String idP = personaje.getFirstChild().getNodeValue();
                personaje = personaje.getNextSibling().getNextSibling();
                PJs.add(idP);
            }
        }
    } catch (Exception e) {
        System.out.println("Error al cargar la lista de personajes en ProfesorSOS");
        e.printStackTrace();
        PJs = new LinkedList();
    }
}

public void guardarLista() {
    try {
        DocumentBuilder builder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
        Document doc = builder.newDocument();
        Element personajes = doc.createElement("Personajes");
        doc.appendChild(personajes);
        int longitud = PJs.size();
        for (int i = 0; i < longitud; i++) {

```

```

        Element personaje = doc.createElement("Personaje");
        personajes.appendChild(personaje);
        personaje.appendChild(doc.createTextNode((String) PJs.get(i)));
    }
    Transformer trans = TransformerFactory.newInstance().newTransformer();
    trans.setOutputProperty(OutputKeys.ENCODING, "ISO-8859-1");
    trans.setOutputProperty(OutputKeys.INDENT, "yes");
    trans.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, "personajes.dtd");
    trans.transform(new DOMSource(doc), new StreamResult(new File(getIdPNJ() + ".xml")));
} catch (Exception e) {
    System.out.println("Error al guardar la lista de personajes en ProfesorSOS");
    e.printStackTrace();
}
}

public void setIdPNJ(String string) {
    super.setIdPNJ(string);
    cargarLista();
}
public void setPJs(LinkedList lista) {
    PJs = lista;
    guardarLista();
}
}

```

Obsérvese que se ha utilizado la funcionalidad extendida descrita en 2.2.5 para diferencia entre los alumnos que ya han ido a la revisión y los que no para darles una respuesta diferente según el caso.

Secretario (C)

Código correspondiente (Secretario.java):

```

package agm.objetos.personajesNoJugadores;

import java.util.LinkedList;
import agm.objetos.ObjetoActualizacion;
import agm.objetos.ObjetoComunicacion;
import agm.objetos.objetosNoPersonaje.Acciones;
import agm.servidor.habitaciones.Habitacion;

public class Secretario extends PersonajeNoJugadorClase {
    public Secretario() {
        tipo = this.getClass().getName();
    }

    public Secretario(String idPNJ, String nombre, String estado, String idHab, int posX, int posY) {
        super(idPNJ, nombre, idHab, posX, posY, estado);
        tipo = this.getClass().getName();
    }

    public void cambiaEstado() {
        if (!estado.equals("AUSENTE")) {
            if (estado.equals("CONVERSANDO")) {
                estado = "ESPERANDO";
            } else {
                estado = "CONVERSANDO";
            }
        } else estado = "ESPERANDO";
    }

    public Acciones ejecutarAccion(String idA, String idO, String idPj) {
        try {
            if ("HABLAR".equals(idA)) {
                if (estado.equals("AUSENTE")) {
                    return new Acciones() {
                        public void ejecutar(String idHab, String idPj) {
                            Habitacion h = conseguirHabitacion(idHab);
                            mostrarResultado(h,

```

```

        generaNotificacion(idPJ,
            "Debi desconfiar, esta ausente, lo que veo solo es la estela que dejo en su huida"));
    }
    public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
        return true;
    };
};
}
if (estado.equals("CONVERSANDO")) { //esta hablando con otro
    return new Acciones() {
        public void ejecutar(String idHab, String idPJ) {
            Habitacion h = conseguirHabitacion(idHab);
            mostrarResultado(h,
                generaNotificacion(idPJ, "Vaya, ahora esta ocupado, esperare un poco, no creo que el que esta
solicitando ayuda tarde mucho en desesperar"));
        }
        public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
            return true;
        }
    };
} else { //esta libre
    Acciones a = teRespondo("OPCION INICIAL", idPJ);
    return a;
}
} else if ("MIRAR".equals(idA)) {
    return new Acciones() {
        public void ejecutar(String idHab, String idPJ) {
            Habitacion h = conseguirHabitacion(idHab);
            ObjetoActualizacion oA = generaNotificacion(idPJ,
                "Cuenta la leyenda que esta para ayudar, pero lo unico cierto es que aumentan la clientela de aspirinas
bayer");
            mostrarResultado(h, oA);
        }
        public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
            return true;
        }
    };
} else {
    return new Acciones() {
        public void ejecutar(String idHab, String idPJ) {
            Habitacion h = conseguirHabitacion(idHab);
            mostrarResultado(h, generaNotificacion(idPJ, "No"));
        }
        public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
            return true;
        }
    };
};
}
} catch (Exception e) {
    System.out.println("Excepcion en ejecutarAccion");
    e.printStackTrace();
    return null;
}
}

public Acciones teRespondo(String opcion, String idPJ) {
    if (estado.equals("ESPERANDO")) estado = "CONVERSANDO";
    if (opcion.equals("OPCION INICIAL")) {
        return new Acciones() {
            public void ejecutar(String idHab, String idPJ) {
                Habitacion h = conseguirHabitacion(idHab);
                ObjetoComunicacion o = new ObjetoComunicacion();
                LinkedList conversacion = new LinkedList();
                o.setEmpiezaJugador(true);
                conversacion.add("Hola");
                conversacion.add("¿Que quieres?");
                o.setConversacionFija(conversacion);
                LinkedList opciones = new LinkedList();
                opciones.add(new Frase("Le estado observando, es increible la labor que esta llevando aqui, debe estar
exhausto", "1S"));
                opciones.add(new Frase("Necesito informacion sobre las becas", "2S"));
                opciones.add(new Frase("Necesito informacion sobre la matricula", "2S"));
                opciones.add(new Frase("Vengo de Yugoslavia, he llegado hoy mismo, me he perdido y he aparecido aqui,
podria ayudarme a llegar a la embajada?", "2S"));
                o.setOpciones(opciones);
            }
        };
    }
}

```

```

        o.setIdPJ(idPJ);
        o.setNombrePNJ("Secretario");
        mostrarResultado(h, o);
    }
    public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
        return true;
    }
};
} else if (opcion.equals("1S")) {
    estado = "AUSENTE";
    return new Acciones() {
        public void ejecutar(String idHab, String idPJ) {
            String idObj = null;
            Habitacion h = null;
            ObjetoComunicacion o = null;
            LinkedList conversacion = null;
            h = conseguirHabitacion(idHab);
            o = new ObjetoComunicacion();
            conversacion = new LinkedList();
            o.setEmpiezaJugador(false);
            conversacion.add("Si, es verdad. Sera mejor que vaya a la cafeteria a descansar un poco, que si no luego no  
rindo y muchos alumnos se verian perjudicados");
            o.setConversacionFija(conversacion);
            o.setIdPJ(idPJ);
            o.setNombrePNJ("Secretario");
            mostrarResultado(h, o);
        }
        public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
            return true;
        }
    };
} else if (opcion.equals("2S")) {
    estado = "AUSENTE";
    return new Acciones() {
        public void ejecutar(String idHab, String idPJ) {
            String idObj = null;
            Habitacion h = null;
            ObjetoComunicacion o = null;
            LinkedList conversacion = null;
            h = conseguirHabitacion(idHab);
            o = new ObjetoComunicacion();
            conversacion = new LinkedList();
            o.setEmpiezaJugador(false);
            conversacion.add("Eso esta en la web");
            conversacion.add("No, ya he mirado ahí;½");
            conversacion.add("Te digo que esta en la web");
            conversacion.add("Lo he mirado 200 veces y no esta");
            conversacion.add("Pues esta en los tableros");
            conversacion.add("No, tampoco esta ahí;½");
            conversacion.add("Tiene que estar");
            conversacion.add("Pues salga a buscarlo a ver si lo encuentra, porque yo llevo dos horas y no he visto nada que  
me sirva de ayuda");
            conversacion.add("Uuuuum... espera aqui un momento, voy a preguntarle a mi compañero a ver si sabe algo");
            o.setConversacionFija(conversacion);
            o.setIdPJ(idPJ);
            o.setNombrePNJ("Secretario");
            mostrarResultado(h, o);
        }
        public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
            return true;
        }
    };
} else return null;
}
}

```

Obsérvese que se ha aprovechado la naturaleza String del atributo ‘Estado’ para añadir un estado mas a los ya habituales, el estado AUSENTE.

Taquilla (TJ)

Código correspondiente (Taquilla.java):

```
package agm.objetos.objetosNoPersonaje;

import java.util.LinkedList;
import agm.objetos.ObjetoActualizacion;
import agm.servidor.habitaciones.Habitacion;

public class Taquilla extends ObjetoNoPersonajeClase {

    public void cambiaEstado() {
        if (estado.equals("CERRADA")) estado = "ABIERTA";
        else estado = "CERRADA";
    }

    public Acciones ejecutarAccion(String idA, String IdOd, String idPJ) {
        try {
            if ("MIRAR".equals(idA)) {
                return new Acciones() {
                    public void ejecutar(String idHab, String idPJ) {
                        Habitacion h = conseguirHabitacion(idHab);
                        ObjetoActualizacion oA = generaNotificacion(idPJ, "Esta es mi taquilla, donde guardo mis cosas");
                        mostrarResultado(h, oA);
                    }
                    public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
                        return true;
                    }
                };
            } else if ("USAR".equals(idA)) {
                String nombre;
                ObjetoNoPersonajeClase oNPC = null;
                if (IdOd != null) {
                    Habitacion h = Acciones.conseguirHabitacion(idHab);
                    oNPC = h.buscarObjeto(IdOd, idPJ);
                    if (oNPC != null) {
                        String tipo = oNPC.getTipo();
                        int ind = tipo.lastIndexOf(",");
                        nombre = tipo.substring(ind + 1);
                    } else nombre = "";
                } else nombre = "";
                if (nombre.equals("Llave taquilla")) {
                    if (estado.equals("CERRADA")) {
                        Acciones a;
                        a = new Acciones() {
                            public void ejecutar(String idHab, String idPJ) {
                                Habitacion h = conseguirHabitacion(idHab);
                                cambiarEstadoObjeto(h, (String) parametros.get(0), idPJ, "Ale, ya esta abierta la taquilla");
                            }
                            public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
                                return true;
                            }
                        };
                    } else {
                        Acciones a;
                        a = new Acciones() {
                            public void ejecutar(String idHab, String idPJ) {
                                Habitacion h = conseguirHabitacion(idHab);
                                cambiarEstadoObjeto(h, (String) parametros.get(0), idPJ, "Ale, ya he cerrado al taquilla");
                            }
                            public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
                                return true;
                            }
                        };
                    }
                }
                LinkedList parametros = new LinkedList();
                parametros.add(getIdO());
                a.setParametros(parametros);
                return a;
            } else {
                Acciones a;
                a = new Acciones() {
                    public void ejecutar(String idHab, String idPJ) {
                        Habitacion h = conseguirHabitacion(idHab);
                        cambiarEstadoObjeto(h, (String) parametros.get(0), idPJ, "Ale, ya he cerrado al taquilla");
                    }
                    public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
                        return true;
                    }
                };
                LinkedList parametros = new LinkedList();
                parametros.add(getIdO());
                a.setParametros(parametros);
                return a;
            }
        } else {
            Acciones a;
        }
    }
}
```



```

        a = new Acciones() {
            public void ejecutar(String idHab, String idPJ) {
                Habitacion h = conseguirHabitacion(idHab);
                ObjetoActualizacion oA = generaNotificacion(idPJ, "No tiene sentido");
                mostrarResultado(h, oA);
            }
            public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
                return true;
            }
        };
        return a;
    }
} else if ("COGER".equals(idA)) {
    if (estado.equals("ABIERTA")) {
        return new Acciones() {
            public void ejecutar(String idHab, String idPJ) {
                Habitacion h = conseguirHabitacion(idHab);
                meterEnInventario(h, idPJ, "agm.objetos.objetosNoPersonaje.CarneJ", "Ya tengo mi documento
identificativo.");
            }
            public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
                return true;
            }
        };
    } else if ((estado.equals("CERRADA"))) {
        return new Acciones() {
            public void ejecutar(String idHab, String idPJ) {
                Habitacion h = conseguirHabitacion(idHab);
                ObjetoActualizacion oA = generaNotificacion(idPJ, "Para coger mis cosas debería primero abrir la
taquilla");
                mostrarResultado(h, oA);
            }
            public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
                return true;
            }
        };
    } else
        return new Acciones() {
            public void ejecutar(String idHab, String idPJ) {
                Habitacion h = conseguirHabitacion(idHab);
                ObjetoActualizacion oA = generaNotificacion(idPJ, "Ya no hay nada mas aqui");
                mostrarResultado(h, oA);
            }
            public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
                return true;
            }
        };
    } else {
        return new Acciones() {
            public void ejecutar(String idHab, String idPJ) {
                Habitacion h = conseguirHabitacion(idHab);
                ObjetoActualizacion oA = generaNotificacion(idPJ, "Eso no serviría para nada");
                mostrarResultado(h, oA);
            }
            public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
                return true;
            }
        };
    }
} catch (Exception e) {
    return null;
}
}

public Taquilla(String idO, String estado, String idHab, int posX, int posY) {
    super(idO, estado, idHab, posX, posY);
    tipo = this.getClass().getName();
}

public Taquilla() {
    tipo = this.getClass().getName();
}
}

```

Llave de la taquilla (LLTJ)

Código correspondiente (Llave_taquilla.java):

```
package agm.objetos.objetosNoPersonaje;

import java.util.LinkedList;
import agm.objetos.ObjetoActualizacion;
import agm.servidor.habitaciones.Habitacion;

public class Llave_taquilla extends ObjetoNoPersonajeClase {
    //Este objeto tiene un unico estado
    public void cambiaEstado() { }

    public Acciones ejecutarAccion(String idA, String IdOd, String idPJ) {
        try {
            if ("MIRAR".equals(idA)) {
                return new Acciones() {
                    public void ejecutar(String idHab, String idPJ) {
                        Habitacion h = conseguirHabitacion(idHab);
                        ObjetoActualizacion oA = generaNotificacion(idPJ, "Esta es la llave que abre mi taquilla del pasillo");
                        mostrarResultado(h, oA);
                    }
                    public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
                        return true;
                    }
                };
            } else if ("USAR".equals(idA)) {
                String nombre;
                ObjetoNoPersonajeClase oNPC = null;
                if (IdOd != null) {
                    HabitacionHome habHome = Acciones.getHabitacionHome();
                    Habitacion h = habHome.findByIdPJ(idPJ);
                    oNPC = h.buscarObjeto(IdOd, idPJ);
                    if (oNPC != null) {
                        String tipo = oNPC.getTipo();
                        int ind = tipo.lastIndexOf(".");
                        nombre = tipo.substring(ind + 1);
                    } else nombre = "";
                } else nombre = "";
                if ((nombre.equals("Taquilla"))) {
                    Acciones a = new Acciones() {
                        public void ejecutar(String idHab, String idPJ) {
                            Habitacion h = conseguirHabitacion(idHab);
                            cambiarEstadoObjeto(h, (String) parametros.get(0), idPJ, "Usando la llave con la taquilla puedo abrirla o cerrarla");
                        }
                        public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
                            return true;
                        }
                    };
                } else {
                    LinkedList parametros = new LinkedList();
                    parametros.add(getIdO());
                    a.setParametros(parametros);
                    return a;
                }
            } else {
                return new Acciones() {
                    public void ejecutar(String idHab, String idPJ) {
                        Habitacion h = conseguirHabitacion(idHab);
                        ObjetoActualizacion oA = generaNotificacion(idPJ, "Eso no servira para nada");
                        mostrarResultado(h, oA);
                    }
                    public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
                        return true;
                    }
                };
            }
        } else {
            return new Acciones() {
                public void ejecutar(String idHab, String idPJ) {
                    Habitacion h = conseguirHabitacion(idHab);
                    ObjetoActualizacion oA = generaNotificacion(idPJ, "No puedo hacer eso");
                    mostrarResultado(h, oA);
                }
            };
        }
    }
}
```

```

    }
    public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
        return true;
    }
};
} catch (Exception e) {
    return null;
}
}

public Llave_taquilla(String idO, String estado, String idHab, int posX, int posY) {
    super(idO, estado, idHab, posX, posY);
    tipo = this.getClass().getName();
}

public Llave_taquilla() {
    estado = "ESTADO";
    tipo = this.getClass().getName();
}
}
}

```

CarneJ (CEJ)

Código correspondiente (CarneJ.java) :

```

package agm.objetos.objetosNoPersonaje;

import agm.objetos.ObjetoActualizacion;
import agm.servidor.habitaciones.Habitacion;

public class CarneJ extends ObjetoNoPersonajeClase {
    //Este objeto tiene un unico estado
    public void cambiaEstado() { }

    public Acciones ejecutarAccion(String idA, String IdOd, String idPJ) {
        if ("MIRAR".equals(idA)) {
            return new Acciones() {
                public void ejecutar(String idHab, String idPJ) {
                    Habitacion h = conseguirHabitacion(idHab);
                    ObjetoActualizacion oA = generaNotificacion(idPJ, "Este es mi carne de estudiante");
                    mostrarResultado(h, oA);
                }
                public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
                    return true;
                }
            };
        } else
            return new Acciones() {
                public void ejecutar(String idHab, String idPJ) {
                    Habitacion h = conseguirHabitacion(idHab);
                    ObjetoActualizacion oA = generaNotificacion(idPJ, "Nah... Eso no serviria para nada");
                    mostrarResultado(h, oA);
                }
                public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
                    return true;
                }
            };
    }

    public CarneJ(String idO, String estado, String idHab, int posX, int posY) {
        super(idO, estado, idHab, posX, posY);
        tipo = this.getClass().getName();
    }

    public CarneJ() {
        tipo = this.getClass().getName();
        estado = "ESTADO";
    }
}

```

Caja de carnes I (CCI)

Código correspondiente (CajaDeCarnes.java):

```
package agm.objetos.objetosNoPersonaje;

import java.io.File;
import java.util.LinkedList;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import agm.objetos.ObjetoActualizacion;
import agm.servidor.habitaciones.Habitacion;

public class CajaDeCarnes extends ObjetoNoPersonajeClase {
    private String idSecretario;
    private LinkedList PJs;
    //Este objeto tiene un unico estado
    public void cambiaEstado() { }

    public Acciones ejecutarAccion(String idA, String IdOd, String IdPj) {
        if ("MIRAR".equals(idA)) {
            return new Acciones() {
                public void ejecutar(String idHab, String idPJ) {
                    Habitacion h = conseguirHabitacion(idHab);
                    ObjetoActualizacion oA = generaNotificacion(idPJ, "Aquí es donde se guardan los carnes perdidos");
                    mostrarResultado(h, oA);
                }
                public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
                    return true;
                }
            };
        } else if ("COGER".equals(idA)) {
            if (!(PJs.contains(IdPj))) {
                return new Acciones() {
                    public void ejecutar(String idHab, String idPJ) {
                        Habitacion h = conseguirHabitacion(idHab);
                        if (interpretaCondicion(h, null, idPJ)) {
                            PJs.add(idPJ);
                            guardarLista();
                            meterEnInventario(h, idPJ, "agm.objetos.objetosNoPersonaje.CarneI", "Ya tengo mi falsa identidad");
                        } else {
                            ObjetoActualizacion oa = generaNotificacion(idPJ, "Debería esperar a que se ausentara el secretario");
                            mostrarResultado(h, oa);
                        }
                    }
                };
            } else {
                return new Acciones() {
                    public void ejecutar(String idHab, String idPJ) {
                        Habitacion h = conseguirHabitacion(idHab);
                        ObjetoActualizacion oa = generaNotificacion(idPJ, "Solo necesito un carnet falso");
                    }
                };
            }
        } else {
            return new Acciones() {
                public void ejecutar(String idHab, String idPJ) {
                    Habitacion h = conseguirHabitacion(idHab);
                    ObjetoActualizacion oa = generaNotificacion(idPJ, "Solo necesito un carnet falso");
                }
            };
        }
    }
}
```

```

        } else {
            oA = generaNotificacion(idPJ, "Deberia esperar a que se ausentara el secretario");
        }
    }
    mostrarResultado(h, oA);
}
public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
    try { //Buscar secretario
        agm.objetos.personajesNoJugadores.PersonajeNoJugadorClase pnj;
        pnj = hab1.buscarPNJEnHabitacion(idSecretario);
        if ((pnj.getEstado().equals("AUSENTE"))) {
            pnj.setEstado("ESPERANDO");
            return true;
        } else return false;
    } catch (Exception e) {
        return false;
    }
}
};
} else
return new Acciones() {
    public void ejecutar(String idHab, String idPJ) {
        Habitacion h = conseguirHabitacion(idHab);
        ObjetoActualizacion oA = generaNotificacion(idPJ, "Nah... Eso no serviria para nada");
        mostrarResultado(h, oA);
    }
    public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
        return true;
    }
};
}

public CajaDeCarnes(String idO, String estado, String idHab, int posX, int posY) {
    super(idO, estado, idHab, posX, posY);
    tipo = this.getClass().getName();
}

public CajaDeCarnes() {
    tipo = this.getClass().getName();
    estado = "ESTADO";
}

public void setSecretario(String s) {
    idSecretario = s;
    guardarSecretario();
}

public void cargarLista() {
    PJs = new LinkedList();
    Document document = null;
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    try {
        DocumentBuilder builder = factory.newDocumentBuilder();
        document = builder.parse(new File(getIdO() + ".xml"));
        PJs = new LinkedList();
        Node raiz = document.getFirstChild().getNextSibling(); //personajes
        if (raiz.getFirstChild() != null) {
            Node personaje = raiz.getFirstChild().getNextSibling(); //personaje
            while (personaje != null) {
                String idP = personaje.getFirstChild().getNodeValue();
                personaje = personaje.getNextSibling().getNextSibling();
                PJs.add(idP);
            }
        }
    } catch (Exception e) {
        System.out.println("Error al cargar la lista de personajes con CajaDeCarnes");
        e.printStackTrace();
        PJs = new LinkedList();
    }
}

public void guardarLista() {
    try {
        DocumentBuilder builder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
        Document doc = builder.newDocument();
    }
}

```

```

        Element personajes = doc.createElement("Personajes");
        doc.appendChild(personajes);
        int longitud = PJs.size();
        for (int i = 0; i < longitud; i++) {
            Element personaje = doc.createElement("Personaje");
            personajes.appendChild(personaje);
            personaje.appendChild(doc.createTextNode((String) PJs.get(i)));
        }
        Transformer trans = TransformerFactory.newInstance().newTransformer();
        trans.setOutputProperty(OutputKeys.ENCODING, "ISO-8859-1");
        trans.setOutputProperty(OutputKeys.INDENT, "yes");
        trans.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, "personajes.dtd");
        trans.transform(new DOMSource(doc), new StreamResult(new File(getIdO() + ".xml")));
    } catch (Exception e) {
        System.out.println("Error al guardar la lista de personajes con CajaDeCarnes");
        e.printStackTrace();
    }
}

public void setIdO(String string) {
    super.setIdO(string);
    cargarLista();
}

public void setPJs(LinkedList lista) {
    PJs = lista;
    guardarLista();
}

public void cargarSecretario() {
    Document document = null;
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    try {
        DocumentBuilder builder = factory.newDocumentBuilder();
        document = builder.parse(new File(getIdO() + ".xml"));
        Node raiz = document.getFirstChild().getNextSibling();
        if (raiz != null) {
            idSecretario = raiz.getFirstChild().getNodeValue();
        }
    } catch (Exception e) {
        System.out.println("Error al cargar el secretario en el CajaDeCarnes");
        e.printStackTrace();
        idSecretario = "";
    }
}

public void guardarSecretario() {
    try {
        DocumentBuilder builder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
        Document doc = builder.newDocument();
        Element secretario = doc.createElement("Secretario");
        doc.appendChild(secretario);
        secretario.appendChild(doc.createTextNode(idSecretario));
        Transformer trans = TransformerFactory.newInstance().newTransformer();
        trans.setOutputProperty(OutputKeys.ENCODING, "ISO-8859-1");
        trans.setOutputProperty(OutputKeys.INDENT, "yes");
        trans.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, "secretario.dtd");
        trans.transform(new DOMSource(doc), new StreamResult(new File(getIdO() + ".xml")));
    } catch (Exception e) {
        System.out.println("Error al guardar el secretario en CajaDeCarnes");
        e.printStackTrace();
    }
}

public String getIdSecretario() {
    return idSecretario;
}

public void setIdsecretario(String string) {
    idSecretario = string;
    guardarSecretario();
}
}

```

Carnel (CEI)

Código correspondiente (CarneI.java):

```
package agm.objetos.objetosNoPersonaje;

import agm.objetos.ObjetoActualizacion;
import agm.servidor.habitaciones.Habitacion;

public class CarneI extends ObjetoNoPersonajeClase {
    //Este objeto tiene un unico estado
    public void cambiaEstado() { }

    public Acciones ejecutarAccion(String idA, String IdOd, String IdPJ) {
        if ("MIRAR".equals(idA)) {
            return new Acciones() {
                public void ejecutar(String idHab, String idPJ) {
                    Habitacion h = conseguirHabitacion(idHab);
                    ObjetoActualizacion oA = generaNotificacion(idPJ, "Este es el carne de estudiante de I, todo un heroe sin saberlo");
                    mostrarResultado(h, oA);
                }
                public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
                    return true;
                }
            };
        } else
            return new Acciones() {
                public void ejecutar(String idHab, String idPJ) {
                    Habitacion h = conseguirHabitacion(idHab);
                    ObjetoActualizacion oA = generaNotificacion(idPJ, "Nah... Eso no serviria para nada");
                    mostrarResultado(h, oA);
                }
                public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
                    return true;
                }
            };
    }

    public CarneI(String idO, String estado, String idHab, int posX, int posY) {
        super(idO, estado, idHab, posX, posY);
        tipo = this.getClass().getName();
    }

    public CarneI() {
        tipo = this.getClass().getName();
        estado = "ESTADO";
    }

    public void setIdO(String string) {
        super.setIdO(string);
    }
}
```

AprobadoSOS

Código correspondiente (AprobadoSOS.java):

```
package agm.objetos.objetosNoPersonaje;

import agm.objetos.ObjetoActualizacion;
import agm.servidor.habitaciones.Habitacion;

public class AprobadoSOS extends ObjetoNoPersonajeClase {

    public void cambiaEstado() {}

    public Acciones ejecutarAccion(String idA, String IdOd, String idPJ) {
        if ("MIRAR".equals(idA)) {
            return new Acciones() {
                public void ejecutar(String idHab, String idPJ) {
```

```

        Habitacion h = conseguirHabitacion(idHab);
        ObjetoActualizacion oA = generaNotificacion(idPJ, "Es el aprobado SOS");
        mostrarResultado(h, oA);
    }
    public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
        return true;
    }
};
} else {
    return new Acciones() {
        public void ejecutar(String idHab, String idPJ) {
            Habitacion h = conseguirHabitacion(idHab);
            ObjetoActualizacion oA = generaNotificacion(idPJ, "No puedo hacer eso");
            mostrarResultado(h, oA);
        }
        public boolean interpretaCondicion(Habitacion hab1, Habitacion hab2, String idPJ) {
            return true;
        }
    };
}
}

public AprobadoSOS(String idO, String estado, String idHab, int posX, int posY) {
    super(idO, estado, idHab, posX, posY);
    tipo = this.getClass().getName();
}

public AprobadoSOS() {
    tipo = this.getClass().getName();
    estado = "ESTADO";
}
}

```

2.3.5. Remodelación del mapa y ubicación de los objetos y personajes en él

Una vez creados los personajes hay que ubicarlos en alguna habitación del mapa para que puedan ser encontrados por el jugador. Así mismo se actuara con los objetos a excepción de que se trate de un objeto que se vera incluido en el inventario por defecto del jugador.

Para implementar el mapa se procede como se describe en la sección 2.2.6 dándonos el siguiente resultado para construir el mapa que tenemos actualmente y en el cual funciona la aventura implementada en este ejemplo:

```

public void creaMundo() {
    String idHab = null;
    String nombre = null;
    boolean luz = true;
    int ancho = -1;
    int alto = -1;
    java.lang.String mascara = null;
    String idHabDest = null;
    int posX = -1;
    int posY = -1;
    String tipo = null;
    String estado = null;
    java.util.Collection puertas = null;
    Habitacion habitacion = null;
    try {
        String[] tiposParametros = { "java.lang.String" };
        Object[] valorParametros = { "CONFIRMACION" };
        invocaServicioMBean("jboss.mq:service=DestinationManager", "createTopic", tiposParametros, valorParametros);
        String[] tiposParametros2 = { };
        Object[] valorParametros2 = { };
        invocaServicioMBean("jboss.mq:destination:service=Topic,name=CONFIRMACION", "removeAllMessages",
            tiposParametros2, valorParametros2);
    }
}

```



```

String[] tiposParametros3 = {};
Object[] valorParametros3 = {};
invocaServicioMBean( "jboss.mq.destination:service=Queue,name=COLA_KERNEL", "removeAllMessages",
tiposParametros2, valorParametros2);
contObjetos = 1;
contPNJ = 1;
setReferInteg(false);
kernel = kernelHome.create();

crearCasa();
crearHall();
crearAseo();
crearPasillo1();
crearConserjeria();
crearDespachoX();
crearSecretaria();
crearPasillo2();
crearDespachoSOS();
} catch (Exception e) {
    System.out.println("Excepcion en creaMundo.");
    e.printStackTrace();
}
}

private void crearSecretaria() {
    String idHab = "H7";
    String nombre = "Secretaria";
    boolean luz = true;
    int ancho = 12;
    int alto = 7;
    String mascara = "0.238-970.523";
    LinkedList objetos = new LinkedList();
    LinkedList pnjs = new LinkedList();
    int posX = 300;
    int posY = 200;
    String estado = "ESPERANDO";
    String idPNJ = "PNJ" + contPNJ++;
    String nombreO = "Secretario";
    pnjs.add(new Secretario(idPNJ, nombreO, estado, idHab, posX, posY));
    posX = 20;
    posY = 10;
    estado = "ESTADO";
    String idO = "O" + contObjetos++;
    CajaDeCarne ci = new CajaDeCarne(idO, estado, idHab, posX, posY);
    ci.setSecretario(idPNJ);
    ci.setPJs(new LinkedList());
    objetos.add(ci);
    LinkedList puertas = new LinkedList();
    String id0 = "0" + contObjetos++;
    puertas.add(new Puerta(id0, idHab, "H2", 700, 400, "ABIERTA"));
    crearHabitacion(idHab, nombre, luz, ancho, alto, mascara, puertas, objetos, pnjs);
}

private void crearHall() {
    String idHab = "H2";
    LinkedList puertas = new LinkedList();
    String idHabDest = "H3";
    int posX = 1;
    int posY = 400;
    String estado = "CERRADA";
    String idO = "O" + contObjetos++;
    puertas.add(new Puerta(idO, idHab, idHabDest, posX, posY, estado));//puerta al baño
    idHabDest = "H8";
    posX = 850;
    posY = 400;
    estado = "ABIERTA";
    idO = "O" + contObjetos++;
    puertas.add(new Puerta(idO, idHab, idHabDest, posX, posY, estado));//puerta al pasillo
    String nombre = "HALL";
    boolean luz = true;
    int ancho = 12;
    int alto = 7;
    String mascara = "0.400-740.520";
    LinkedList objetos = new LinkedList();
    LinkedList pnjs = new LinkedList();

```

```

posX = 300;
posY = 200;
estado = "ESPERANDO";
String idPNJ = "PNJ" + contPNJ++;
String nombreO = "Conserje";
pnjs.add(new Conserje(idPNJ, nombreO, estado, idHab, posX, posY));
idHabDest = "H6";
posX = 800;
posY = 100;
estado = "CERRADA";
idO = "O" + contObjetos++;
PuertaAConserjeria p = new PuertaAConserjeria(idO, estado, idHab, idHabDest, posX, posY);
p.setIdConserje(idPNJ);
puertas.add(p); //puerta a Conserjeria
idHabDest = "H7";
posX = 200;
posY = 100;
estado = "ABIERTA";
idO = "O" + contObjetos++;
Puerta ps = new Puerta(idO, idHab, idHabDest, posX, posY, "ABIERTA");
puertas.add(ps); //puerta a Secretaria
crearHabitacion(idHab, nombre, luz, ancho, alto, mascara, puertas, objetos, pnjs);
try {
    HabitacionHome habitacionHome = HabitacionUtil.getHome();
    Habitacion h = habitacionHome.findByPrimaryKey(new HabitacionPK(idHab));
    System.out.println("Habitacion " + idHab + "; " + h.getMascara());
} catch (Exception e) {
    System.out.println("Excepcion al crear Hall");
}
}

private void crearPasillo1() {
    String idHab = "H8";
    String idHabDest = "H2";
    int posX = 870;
    int posY = 200;
    String estado = "ABIERTA";
    String idO = "O" + contObjetos++;
    LinkedList puertas = new LinkedList();
    puertas.add(new Puerta(idO, idHab, idHabDest, posX, posY, estado)); //Puerta al primer pasillo
    idHabDest = "H4";
    posX = 1;
    posY = 400;
    estado = "ABIERTA";
    idO = "O" + contObjetos++;
    puertas.add(new Puerta(idO, idHab, idHabDest, posX, posY, estado)); //puerta al Hall
    String nombre = "PASILLO1";
    boolean luz = true;
    int ancho = 12;
    int alto = 7;
    String mascara = "0.480-970.525";
    LinkedList objetos = new LinkedList();
    LinkedList pnjs = new LinkedList();
    posX = 20;
    posY = 10;
    estado = "CERRADA";
    idO = "O" + contObjetos++;
    Taquilla ci = new Taquilla(idO, estado, idHab, posX, posY);
    objetos.add(ci);
    crearHabitacion(idHab, nombre, luz, ancho, alto, mascara, puertas, objetos, pnjs);
}

private void crearDespachoSOS() {
    String idHab = "H9";
    String idHabDest = "H4";
    int posX = 850;
    int posY = 400;
    String estado = "ABIERTA";
    String idO = "O" + contObjetos++;
    LinkedList puertas = new LinkedList();
    puertas.add(new Puerta(idO, idHab, idHabDest, posX, posY, estado)); //puerta al pasillo
    String nombre = "DESPACHO2";
    boolean luz = true;
    int ancho = 12;
    int alto = 7;

```

```

String mascara = "250.370-835.520";
LinkedList objetos = new LinkedList();
LinkedList pnjs = new LinkedList();
posX = 170;
posY = 300;
estado = "ESPERANDO";
String idPNJ = "PNJ" + contPNJ++;
String nombreO = "Profesor SOS";
ProfesorSOS p = new ProfesorSOS(idPNJ, nombreO, estado, idHab, posX, posY);
pnjs.add(p);
crearHabitacion(idHab, nombre, luz, ancho, alto, mascara, puertas, objetos, pnjs);
}

```

2.3.6. Definir los objetos del inventario por defecto

Ahora solo nos queda por incluir en el inventario por defecto siguiendo el procedimiento descrito en la sección 2.2.7 los nuevos objetos de los que debe disponer el jugador tras su creación.

Escribimos en la función objetosIniciales() de KernelBean.java:

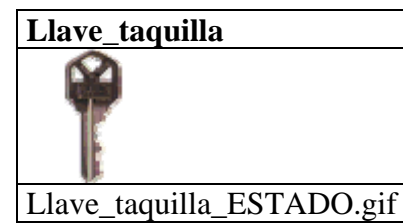
```

public LinkedList objetosIniciales() {
    LinkedList ob = new LinkedList();
    Llave_taquilla lt = new Llave_taquilla();
    lt.setIdO(KernelUtil.generateGUID(It));
    ob.add(lt);
    return ob;
}



```




2.3.7. Representación grafica de los objetos y personajes

La representación grafica de cada objeto vendrá dada como se indica en la sección 2.2.8 de manera que en el directorio \src\agm\clienteJugador\interfazGrafica\imagenes se incluyen las siguientes imágenes:



- Taquilla_ABIERTA.gif (imagen vacía porque el propio fondo de la habitación vale)
- Taquilla_CERRADA.gif (imagen vacía porque el propio fondo de la habitación vale)

ProfesorSOS	
	
ProfesorSOS_CONVERSANDO.png	ProfesorSOS_ESPERANDO.png

Secretario		
		
Secretario_AUSENTE.png	Secretario_CONVERSANDO.png	Secretario_CONVERSANDO.png



2.3.8. *Ensamblado del código generado*

Una vez generado todo el código nuevo y elegido las imágenes que representan a los objetos queda por colocar los archivos para que se integren en la AGM.

La disposición de los archivos es la siguiente:

Paquete **agm.objetos.personajesNoJugadores:**

- ProfesorSOS.java
- Secretario.java

Paquete **agm.objetos.objetosNoPersonaje:**

- Taquilla.java
- Llave_taquilla.java
- CarneJ.java
- CajaDeCarnes.java
- CarneI.java
- AprobadoSOS.java

Paquete **agm.servidor.kernel:**

- GestorSistemaBean.java (versión con el nuevo método creaMundo() y las nuevas funciones para las nuevas habitaciones)
- KernelBean.java (versión con la modificación del método objetosIniciales() para generar el inventario por defecto)

Paquete **agm.clienteJugador.interfazGrafica.imágenes:**

- CarneI_ESTADO.jpg
- CarneJ_ESTADO.jpg
- AprobadoSOS_ESTADO.jpg
- ProfesorSOS_CONVERSANDO.png
- ProfesorSOS_ESPERANDO.png
- Secretario_AUSENTE.png
- Secretario_CONVERSANDO.png
- Secretario_ESPERANDO.png
- Taquilla_ABIERTA.gif
- Taquilla_CERRADA.gif
- Llave_taquilla_ESTADO.gif
- CajaDeCarnes_ESTADO.gif
- H2.jpg
- H9.jpg
- H7.jpg
- H8.jpg

3. Pruebas XML

3.1. Introducción

La siguiente documentación muestra el diseño, desarrollo y uso del paquete de pruebas pruebasXML dentro del desarrollo de la Aventura Gráfica Multi-jugador (desde ahora AGM).

El objetivo era conseguir diseñar un módulo flexible que permitiera la ejecución de diferentes pruebas sobre el AGM mostrando así su comportamiento bajo determinadas circunstancias.

Lo fundamental consiste en el desarrollo de un lenguaje XML que sirva para detallar el conjunto de acciones dentro del juego que constituyen una prueba. De este modo podremos en el futuro codificar distintas pruebas según nuestros intereses.

Una vez definida la prueba o pruebas hay que procesar el documento XML para sacar de él la información que sea de utilidad para el buen funcionamiento del test.

Finalmente, y he aquí el punto más escabroso, hay que conseguir que las acciones que hemos indicado en el documento XML para la prueba sean ejecutadas de forma sincronizada y siguiendo las “reglas del juego”. Es decir, está claro que no es posible conectar al juego a un personaje X sin que este haya sido creado anteriormente, ni podemos hablar con un personaje no jugador que no se encuentre en la habitación en la que nosotros nos encontramos, por poner dos ejemplos significativos.

3.2. Definición del lenguaje XML

3.2.1. Especificación del lenguaje con XML Schema

Para la especificación del lenguaje XML empleado para la codificación de las pruebas se ha definido su estructura mediante un archivo de tipo xsd, de modo que todas las pruebas que hagamos tienen que seguir dicha estructura.

En un documento con extensión xsd se define la apariencia de un conjunto de uno o más documentos XML; esto es, que elementos y atributos pueden contener y en qué orden y cuál puede ser su contenido.

Este objetivo de determinar claramente los XML podría haberse llevado a cabo también empleando DTDs, pero al tener estos últimos ciertos inconvenientes respecto a los esquemas escritos con XML Schema, se optó por no usarlos.

XML Schema permite definir elementos locales y globales, además de tener un sistema de tipos de datos, que permite especificar, por ejemplo, si un elemento debería contener un entero o una cadena, etc. De modo, que usar XML Schema permite tener más control sobre los contenidos de un documento XML del que podríamos obtener empleando DTDs.

3.2.2. Pruebas.xsd

Este archivo es el que indica la estructura de los documentos XML que emplearemos para las pruebas.

Su estructura es la que se muestra a continuación:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">

  <xsd:annotation>
  <xsd:documentation>
```

```

Este esquema se usará para validar las pruebas en formato XML del proyecto AGM.
</xsd:documentation>
</xsd:annotation>

<xsd:element name="pruebas" type="tipoPruebas"/>

<xsd:complexType name="tipoPruebas">
  <xsd:attribute name="numPruebas" type="xsd:positiveInteger">
  <xsd:sequence minOccurs="1" maxOccurs="unbounded">
    <xsd:element name="prueba" type="tipoPrueba"/>

    <xsd:complexType name="tipoPrueba">
      <xsd:attribute name="nombre" type="xsd:string">
      <xsd:element name="numPJs" type="xsd:positiveInteger"/>
      <xsd:element name="acciones" type="tipoAcciones"/>

      <xsd:complexType name="tipoAcciones">
        <xsd:sequence minOccurs="1" maxOccurs="unbounded">
          <xsd:element name="accion" type="tipoAccion"/>

          <xsd:complexType name="tipoAccion">
            <xsd:attribute name="tipo" type="xsd:string" use="required">
            <xsd:attribute name="parametro" type="xsd:string">
            <xsd:attribute name="opciones" type="xsd:string">
            </xsd:complexType>

          </xsd:sequence>
        </xsd:complexType>

      </xsd:complexType>

    </xsd:sequence>
  </xsd:complexType>

</xsd:sequence>
</xsd:complexType>

</xsd:schema>

```

De la estructura anterior se pueden sacar las siguientes conclusiones sobre la forma de los documentos XML empleados:

- Un documento XML para pruebas estará formado por un solo elemento de *pruebas* (que identificará a la raíz del documento) de tipo *tipoPruebas*.
- En el elemento *pruebas* hemos de detallar el número de pruebas incluidas, es decir, si hay diferentes listas de acciones para jugadores distintos. Esto se lleva a cabo con el atributo *numPruebas*.
- Dentro del elemento *pruebas* podremos tener 1 o más elementos de tipo *prueba*.
- El elemento *prueba* de tipo *tipoPrueba* especifica un listado de acciones a efectuar con tantos personajes jugadores como se especifique en el elemento *numPJs*.
- El elemento *acciones* de tipo *tipoAcciones* define un listado de acciones definidas por distintos elementos *acción* de tipo *tipoAcción*.
- El tipo *tipoAcción* determina que toda acción estará determinada por tres atributos: *tipo* (que es requerido siempre) y *parámetro* y *opciones*, los cuales son opcionales.
- El atributo *tipo* es de tipo *xsd:string* y define el tipo de las acciones que podemos llegar a realizar: SOLICITUD CREACIÓN, SOLICITUD CONEXIÓN, IR A, COGER, HABLAR, MIRAR y USAR.
- El atributo *parámetro* que es de tipo *xsd:string* define los elementos sobre los que podremos efectuar las acciones antes citadas. De este modo, con las acciones de tipo SOLICITUD CREACIÓN y SOLICITUD CONEXIÓN no pondremos ningún parámetro asociado, pues no lo tienen.

- El atributo *opciones* que es de tipo *xsd:string* define las opciones elegidas en una conversación con un personaje no jugador. Así que solo tendrá sentido para acciones de tipo HABLAR.

Para que los documentos XML sigan esta estructura hay que declararlo explícitamente. Para ello usaremos la siguiente expresión:

```
<pruebas xmlns="pruebas.xsd">
```

3.2.3. Plantilla para pruebas

Se debe seguir la siguiente plantilla a la hora de definir nuevas pruebas que sigan la estructura anterior:

```
<?xml version="1.0"?>
<pruebas xmlns="pruebas.xsd">
  AÑADIR UNO O MÁS ELEMENTOS PRUEBA
  <prueba nombre= INTRODUCIR NOMBRE DE LA PRUEBA DESARROLLADA>
<numPJs> INTRODUCIR NÚMERO DE PERSONAJES JUGADORES QUE VAN A EFECTUAR ESTA PRUEBA
</numPJs>
    <acciones>
      AÑADIR UNO O MÁS ELEMENTOS ACCION
      <accion tipo= parámetro= opciones= />
      .
      .
      .
    </acciones>
  </prueba>
  .
  .
  .
</pruebas>
```

3.3. Procesamiento de documentos XML

Para el procesamiento de documentos XML se usa un parser XML que lee el texto diferenciando entre tags, atributos y textos. En este caso se ha usado el parser XercesJ 2.6 de apache (<http://xml.apache.org>).

El procesamiento se puede hacer con DOM o con SAX. Nosotros nos hemos decantado por usar DOM, debido a que es más fácil.

3.3.1. DOM

El procesamiento con DOM consiste en generar el árbol que representa el documento y almacenarlo por completo en memoria. Una vez en memoria, se procesa como se desee. El fichero XML se traduce en un objeto de tipo Document, el cual contiene nodos. Cada nodo de Document representa a un tag del fichero XML. Un nodo puede contener otros nodos y tener atributos. También, se puede dar el caso de tener nodos de tipo texto, es decir, que no contienen otros nodos, sólo texto plano.

Para realizar el parking DOM hay que seguir las siguientes directrices:

- Hay que inicializar el parser con el fichero a leer.

```
DOMParser parser = new DOMParser();
FileInputStream pcj = new FileInputStream(interfaz.darResp());
InputSource is = new InputSource(pcpj);
parser.parse(is);
doc = parser.getDocument();
```

- Hay que dirigirse al nodo a partir del cual queremos iniciar el procesamiento.

```
NodeList pruebasEnDocumento = doc.getElementsByTagName("pruebas");
Node misPruebas = pruebasEnDocumento.item(0);
```

- Hay que estudiar los nodos que nos interesen obteniendo sus nombres y atributos, así como los tags que contienen.

```
System.out.println("nombre:" + misPruebas.getNodeName());
//Coger el atributo numPruebas de pruebas si es que existe
NamedNodeMap numPruebas = misPruebas.getAttributes();
for (int n = 0; n < numPruebas.getLength(); n++) {
    if (numPruebas.item(n).getNodeName().toLowerCase().equals("numpruebas"))
    {
        System.out.println("numPruebas: " + numPruebas.item(n).getNodeValue());
        try {
            numPru = Integer.parseInt((numPruebas.item(n).getNodeValue()).trim());
            System.out.println(numPru);
        }
        catch (NumberFormatException e) {}
    }
}
```

Las ventajas de hacer este procesamiento son varias, pero fundamentalmente está el hecho de que si se produce alguna modificación en el documento XML (que no afecte gravemente a su estructura) no es necesario hacer cambios en el procesamiento y si el documento a procesar tiene algún tipo de error, este será notificado.

3.3.2. *Requisitos*

Para hacer el procesamiento con DOM o SAX necesitamos estar en disposición de las siguientes librerías:

xercesImpl.jar
xml-apis.jar
xmlParserAPIs.jar

Estas librerías no vienen por defecto con Eclipse así que hay que importarlas. Lo mismo sucede con el JCreator o con el JBuilder.

Una vez importadas hay que hacer uso de las mismas en la clase de nuestra aplicación que vaya a ocuparse del procesamiento de los documentos XML, para ello hay que hacer las inclusiones pertinentes:

```
import org.w3c.dom.*;
import org.w3c.dom.Document;
import org.w3c.dom.DOMImplementation;
import org.apache.xerces.parsers.DOMParser;
import org.xml.sax.InputSource;
```

3.4. *Diseño del paquete de pruebas*

3.4.1. *Diagrama de casos de uso*

Este módulo tiene como única funcionalidad permitir el testeo del AGM mediante la ejecución de pruebas que están especificadas en XML. De esto se explica que el único caso de uso del paquete sea la ejecución de pruebas:

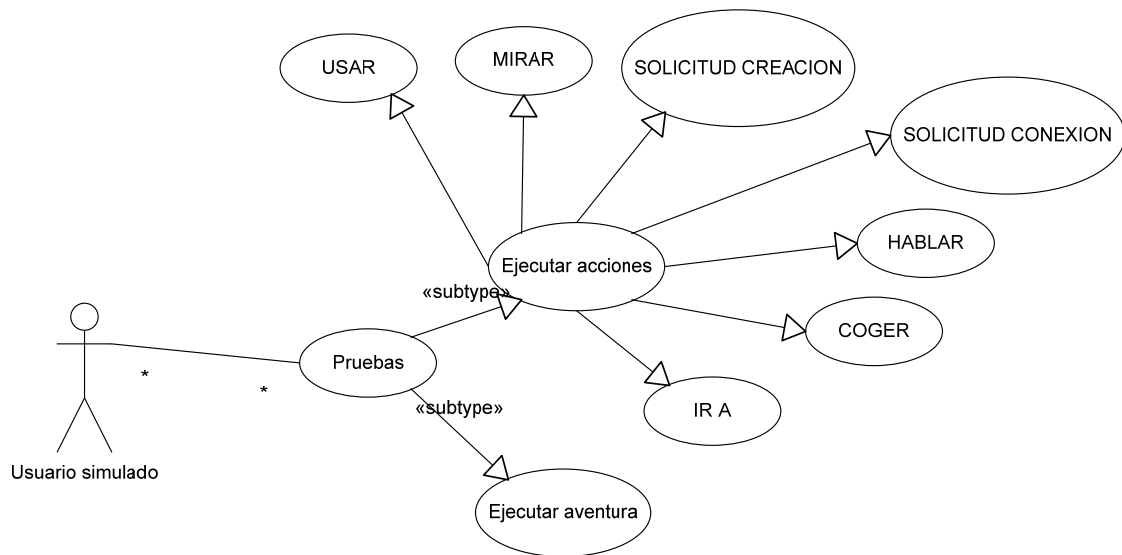


Ilustración 3. Diagrama de casos de uso

3.4.2. Diagrama de actividades

El siguiente diagrama de actividades muestra los pasos que se siguen para el caso de uso anterior:

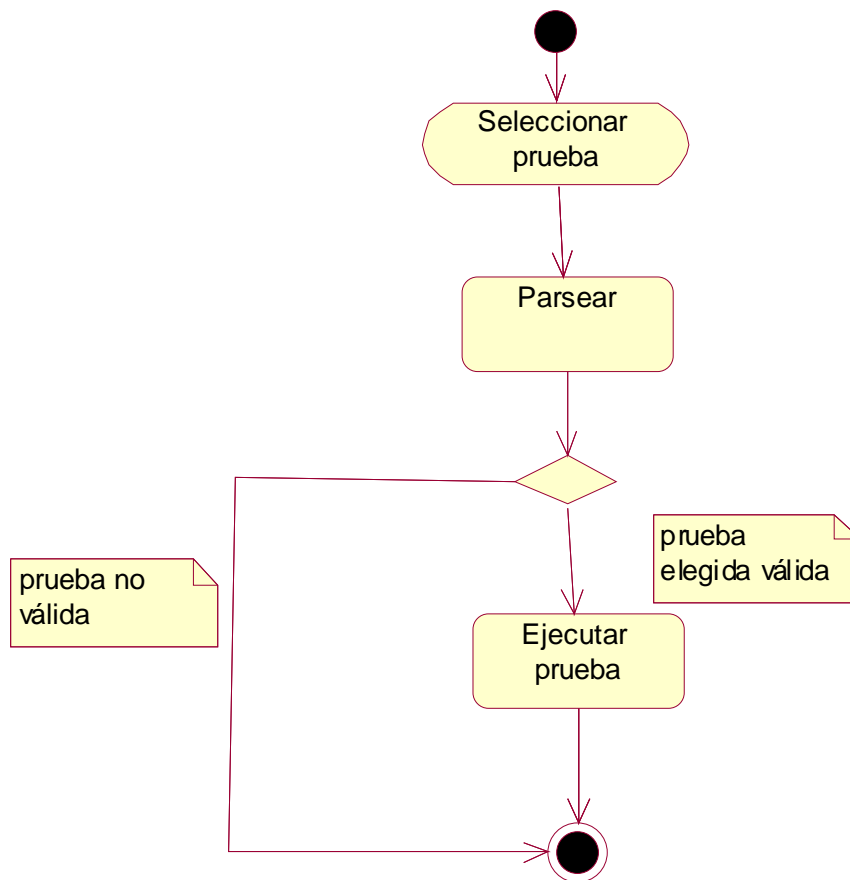


Ilustración 4. Diagrama de actividades

3.4.3. Diagrama de secuencia

El siguiente diagrama de secuencia muestra el flujo de eventos que se producen durante la ejecución de una prueba desde el momento en que se selecciona dicha prueba y se da al botón de aceptar en la interfaz gráfica hasta que se termina la ejecución de dicha prueba, es decir, cuando ya se han realizado todas las acciones especificadas en la prueba.

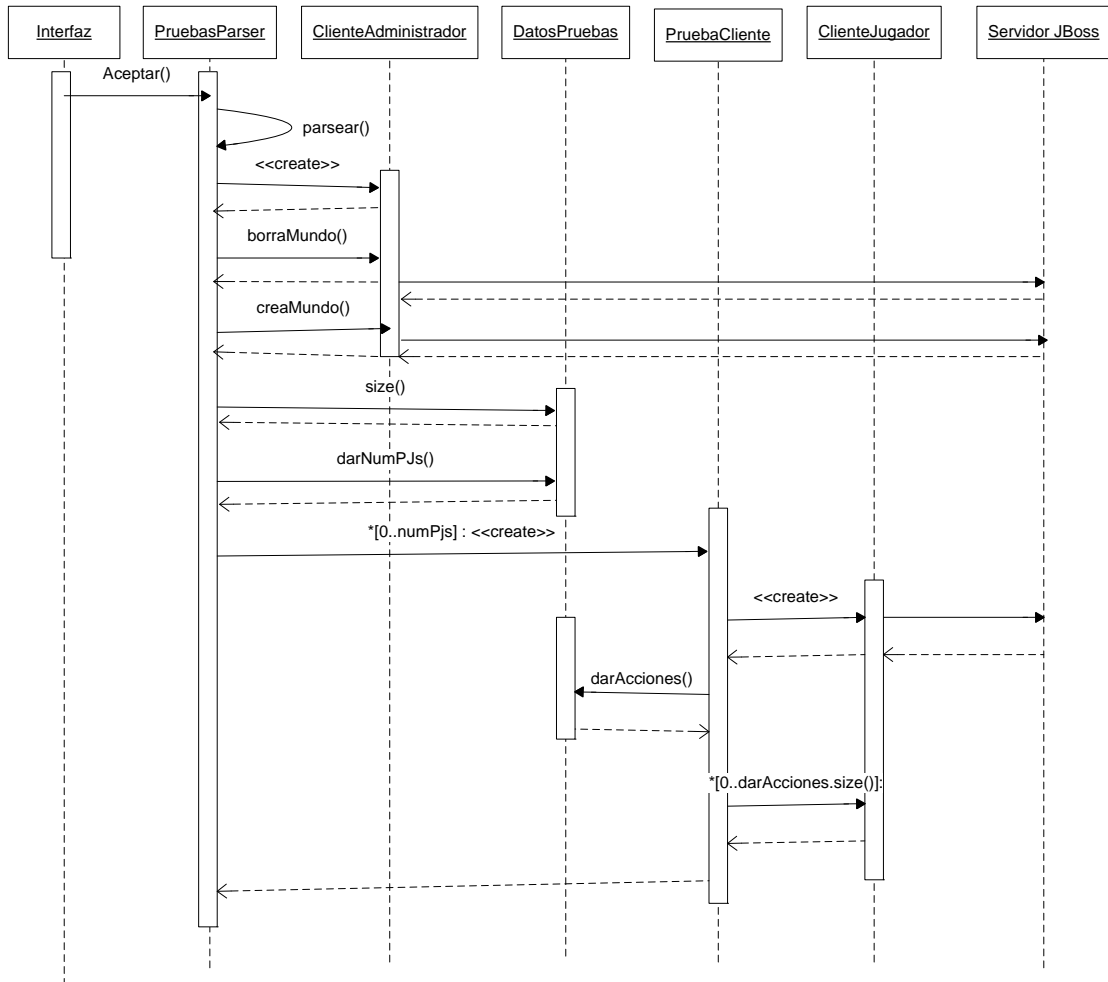


Ilustración 5. Diagrama de secuencia

3.4.4. Paquetes

El paquete pruebasxml forma parte de la aplicación AGM y hace uso de otros paquetes dentro de dicha aplicación para poder efectuar las llamadas necesarias para poder mostrar y probar el funcionamiento correcto del juego bajo diversas circunstancias.

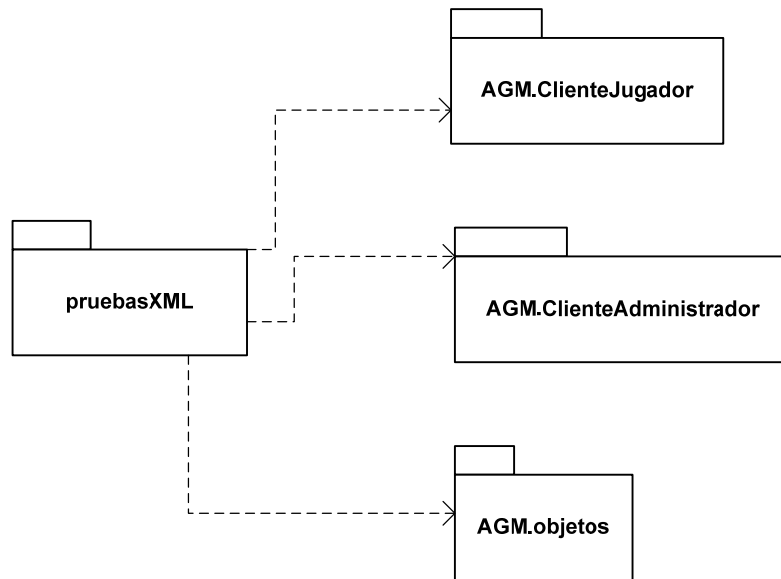


Ilustración 6. Diagrama de paquetes

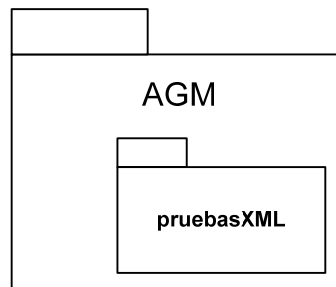


Ilustración 7. Diagrama de paquetes

3.5. Arquitectura del paquete

3.5.1. Diagrama de clases

En el siguiente diagrama de clases se ve las distintas dependencias entre estas clases y las clases ClienteJugador y ClienteAdministrador que aunque no pertenecen al paquete son necesarias llamadas directas a funciones de estas clases para comprobar el funcionamiento del AGM.

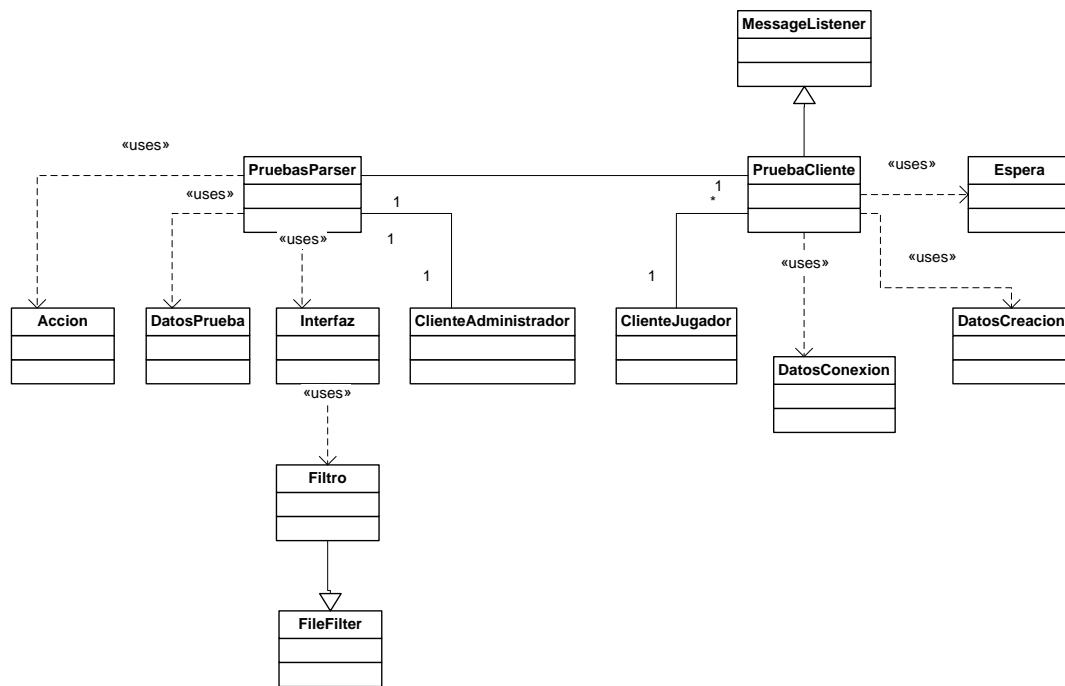


Ilustración 8. Diagrama de clases

3.5.2. *PruebasParser*

Esta clase es la que contiene el main() de la aplicación de prueba. Invoca a la interfaz gráfica de las pruebas (para poder elegir de forma gráfica una prueba del catálogo disponible). Parsea usando un parser DOM la prueba especificada y según la prueba crea tantas instancias de la clase PruebaClientes como sean necesarias. Al parsear el documento XML, introduce los datos que sean de relevancia en un objeto de tipo DatosPrueba, de forma que luego puedan ser accedidos por los clientes para ejecutar las acciones indicadas en la prueba.

También desde aquí se hacen las inicializaciones pertinentes del mundo del juego mediante las llamadas necesarias a ClienteAdministrador.

Atributos

- static Document doc
Representa la traducción del documento XML una vez procesado.
- static Vector dp
Guarda la información de la prueba procesada.
- static Interfaz interfaz
Instancia de la interfaz gráfica de la aplicación.

Métodos

- public static void main(String args[])
Programa principal de la aplicación que llama a la interfaz de usuario para que esta sea visible al usuario; procesa el documento especificado para obtener sus

datos; inicializa al mundo del juego y crea tantas instancias de PruebaCliente como sean necesarias para la realización de la prueba especificada.

- `public static Vector parsear()`
Parsea el documento XML especificado. Llama a `parseo1` o `parseo2` dependiendo de la necesidad que tenga el documento.
- `private static void parseo2(Node actual2)`
Parsea ficheros XML con varias pruebas en su interior, es decir, varios jugadores con distintas acciones cada uno.
- `private static void parseo1(Node misPruebas)`
Parsea ficheros XML con una única prueba en su interior. Es decir, mismas acciones para todos los jugadores implicados en la prueba.

3.5.3. *PruebaCliente*

Representa a ClienteJugador dentro del paquete, es decir hay tantas instancias de PruebaClientes como instancias de ClienteJugador sean necesarias para realizar la prueba especificada, en otras palabras existe una relación de 1 a 1 entre estas dos clases. Tomando los datos de la instancia de DatosPrueba pasados por PruebasParser ejecuta las acciones que vengan indicadas mediante llamadas a los métodos disponibles para tal fin en la clase ClienteJugador.

Nota: El ClienteJugador que se usa tiene un atributo privado de PruebaCliente. Este atributo se le pasa desde PruebaCliente al ser creado el objeto ClienteJugador, para lo cual se ha implementado un nuevo constructor al margen del que ya existía, que recibe como parámetro un objeto de tipo PruebaCliente. De modo que existe una asociación entre ambas clases que permite la navegación de una a otra. Esto es necesario por que se necesita manejar los mensajes recibidos desde el servidor dentro de la clase PruebaCliente (método `onMessage(...)`) pero cuando hacemos llamadas a métodos de ClienteJugador internamente se van produciendo nuevos mensajes que se deben tener en cuenta y que si no se procede como se ha descrito son manejados por el método `onMessage(...)` de ClienteJugador, con lo cual no le llegan a PruebaCliente.

Atributos

- `private String mensajeRecibido`
Tipo de mensaje que recibimos del servidor. Inicialmente vacío.
- `private String mensajeRecibidoObjetosRefresco`
Inicialmente vacío, se rellenará convenientemente cuando recibamos del servidor un mensaje de Objetos Refresco.
- `private String mensajeRecibidoNotificacionTextual`
Inicialmente vacío, se rellenará convenientemente cuando recibamos del servidor un mensaje de Notificación Textual.
- `public static String PersonajeCreado`
Cadena para la comparación con `mensajeRecibido`.
- `public static String PersonajeConectado`
Cadena para la comparación con `mensajeRecibido`.

- `public static String ObjetosRefresco`
Cadena para la comparación con `mensajeRecibido`.
- `public static String NotificacionTextual`
Cadena para la comparación con `mensajeRecibido`.
- `public static String Comunicacion`
Cadena para la comparación con `mensajeRecibido`.
- `private int contador`
Lleva la cuenta del número de mensajes recibidos por las acciones de tipo IR A.
- `private ClienteJugador cj`
Instancia de `ClienteJugador`. Cada Prueba Cliente representa a un `ClienteJugador`.
- `Espera esp`
Objeto para la sincronización entre las acciones.
- `private Message message`
Mensaje recibido del servidor.

Métodos

- `public PruebaCliente(int indice, DatosPrueba dp)`
Constructor de la clase.
- `private void partir(String ops)`
Parte la cadena pasada como parámetro en tantas partes como sea necesario. Se usa para separar las distintas opciones de una conversación con un personaje no jugador.
- `public void onMessage(Message arg0)`
Recibe los mensajes del servidor.
- `public void procesa(javax.jms.Message message)`
Procesa los mensajes del servidor.

3.5.4. DatosPrueba

Encapsula los datos procedentes del fichero XML que se ha procesado.

Atributos

- `private Vector acciones`
Almacena la secuencia de acciones a realizar en una prueba.
- `private int numPJs`
Número de personajes jugadores implicados en la prueba y que, por tanto, han de realizar todas las acciones especificadas en el atributo `acciones`.

Métodos

- `public DatosPrueba(Vector a, int n)`
Constructor con parámetros de la clase. Inicializa los atributos de la clase con los parámetros pasados al constructor.
- `public DatosPrueba()`
Constructor sin parámetros de la clase. Inicializa por defecto los atributos de la clase.
- `public Vector darAcciones()`
Devuelve el atributo acciones.
- `public int darNumPJs()`
Devuelve el atributo numPJs.
- `public void meterAcciones(Vector a)`
Asigna al atributo acciones el parámetro pasado.
- `public void meterNumPJs(int n)`
Asigna al atributo numPJs el parámetro pasado.
- `public void mostrarDatos()`
Muestra los datos encapsulados en el objeto.

3.5.5. Interfaz

Genera la interfaz gráfica necesaria para que el usuario de la aplicación pueda elegir entre las pruebas en formato xml que se encuentran en el directorio de trabajo actual. Realmente nos presenta para elegir entre todos los ficheros de con extensión .xml que tengamos en dicho directorio de trabajo, de modo que si tenemos algún archivo con dicha extensión pero que no represente a una prueba instanciable también se verá aquí, aunque si la elegimos no sea ejecutable.

Atributos

- `private String respuesta`
Representa la prueba seleccionada.

Métodos

- `public Interfaz()`
Constructor de la clase.
- `public String darResp()`
Devuelve el valor que tiene el atributo respuesta para que otras clases puedan hacer uso del mismo.

3.5.6. DatosConexión

Encapsula los datos imprescindibles para efectuar la conexión de un personaje jugador al juego, como son el identificador del jugador y la clave de acceso a la aplicación.

Atributos

- `private String password`
Cadena que representa la contraseña del personaje jugador para conectarse al juego.
- `private String id`
Cadena que representa el identificador del personaje jugador para conectarse al juego.

Métodos

- `public DatosConexion(String id, String password)`
Constructor de la clase que inicializa los atributos de la clase con los pasados como parámetros.
- `public String getId()`
Devuelve el valor del atributo id.
- `public String getPassword()`
Devuelve el valor del atributo password.
- `public void setId(String string)`
Asigna al atributo id el parámetro pasado.
- `public void setPassword(String string)`
Asigna al atributo password el parámetro pasado.

3.5.7. DatosCreación

Encapsula los datos imprescindibles para efectuar la creación de un personaje jugador en el juego, como son el identificador del jugador, la clave de acceso a la aplicación, el nombre del personaje y su imagen.

Atributos

- `private String nombre`
Cadena que representa el nombre del personaje jugador.
- `private String imagen`
Cadena que representa la imagen del personaje jugador.
- `private String password`
Cadena que representa la contraseña del personaje jugador.
- `private String id`
Cadena que representa el identificador del personaje jugador.

Métodos

- `public DatosCreacion(String id, String password, String nombre, String imagen)`
Constructor de la clase que inicializa los atributos de la clase con los pasados como parámetros.
- `public String getPassword()`

Devuelve el valor del atributo password.

- `public String getId()`
Devuelve el valor del atributo id.
- `public String getNombre()`
Devuelve el valor del atributo nombre.
- `public String getImagen()`
Devuelve el valor del atributo imagen.

3.5.8. *Espera*

Clase cuyo fin es mantener la sincronización entre los eventos, de manera que no ejecutemos una acción sobre un cliente hasta que no tengamos la seguridad que la acción precedente se haya terminado de ejecutar correctamente. Para esto hacemos uso de los métodos de sincronización `wait()` y `notify()`.

Atributos

- `private Object obj`
Objeto sobre el que se va a efectuar la sincronización (monitor).

Métodos

- `public Espera()`
Constructor sin parámetros de la clase.
- `synchronized public void espera() throws InterruptedException`
El objeto que efectúe esta llamada quedará en ejecución suspendida hasta que se le despierte.
- `synchronized public void despierta()`
Se despiertan todos los objetos que estuvieran esperando.

3.5.9. *Filtro*

Esta clase extiende a la clase `FileFilter`. Su uso está destinado a filtrar de forma sencilla aquellos ficheros que no cumplen las condiciones que nosotros especifiquemos, en este caso la única condición que imponemos para que `Filtro` admita ciertos ficheros y no otros es que la extensión de los mismos sea `.xml`. Esta clase se utiliza dentro de la clase `Interfaz`.

Métodos

- `public Filtro()`
Constructor sin parámetros de la clase que simplemente efectúa una llamada al constructor de la clase padre (`FileFilter`).
- `public boolean accept(File f)`
Devuelve `true` si el fichero pasado como parámetro es aceptado por el filtro (cumple las restricciones impuestas) y devuelve `false` en caso contrario.
- `public String getDescription()`

Muestra una descripción del filtro, esto es, los ficheros que serán admitidos por él.

3.5.10. *Acción*

Con esta clase encapsulamos los datos referentes a las distintas acciones con se pueden llevar a cabo dentro del AGM (IR A, COGER, HABLAR, CONEXIÓN, CREACIÓN, MIRAR y USAR). Los datos que puede tener una acción son el tipo de acción, el parámetro que representa el objeto o personaje sobre el que se efectúa dicha acción y opciones que representa a parámetros adicionales, como son las opciones de una conversación.

Atributos

- `private String tipo`
Cadena que representa el tipo de acción.
- `private String parametro`
Cadena que representa el parámetro sobre el que se efectúa la acción, si es que es necesario tal parámetro.
- `private String opciones`
Cadena que representa las opciones asociadas a una acción, cuando estas sean necesarias.

Métodos

- `public String darTipo()`
Devuelve el valor del atributo tipo.
- `public String darParametro()`
Devuelve el valor del atributo parámetro.
- `public String darOpciones()`
Devuelve el valor del atributo opciones.
- `public void meterTipo(String tipo)`
Asigna al atributo tipo el parámetro pasado.
- `public void meterParametro(String parametro)`
Asigna al atributo parámetro el parámetro pasado.
- `public void meterOpciones(String opciones)`
Asigna al atributo opciones el parámetro pasado.
- `public Accion()`
Constructor sin parámetros de la clase.
- `public Accion(String s)`
Constructor con parámetros de la clase, inicializa el atributo tipo con el parámetro pasado.

3.6. Tipos de pruebas

Con este entorno de pruebas podemos llevar a cabo distintos tipos de pruebas, como son:

- La creación y conexión de distintos usuarios simulados al sistema.
- Monitorización de lo que sucede en el sistema, teniendo constancia de lo que está sucediendo en cada habitación por medio de la colocación de distintos usuarios simulados en distintas habitaciones.
- Reflejar todas las acciones necesarias para que una aventura se lleve a buen fin y comprobar, por tanto, que el sistema responde como esperamos.

3.7. Tutorial de pruebas

Este apartado tiene la finalidad de documentar e instruir en el proceso de desarrollo de una prueba concreta, bien para probar una aventura completa, bien para probar un comportamiento del sistema.

3.7.1. Componentes involucrados

Antes de escribir una prueba hay que tener claro sobre que objetos y personajes no jugadores vamos a actuar, es decir, que recursos del sistema vamos a emplear.

Cada objeto y cada personaje no jugador viene identificado por un identificador que se puede consultar en la base de datos de la aplicación. Por comodidad, en los siguientes apartados mostraremos cuales son dichos identificadores para el estado actual del AGM.

Puertas

Hay muchos objetos de este tipo en la aplicación, pues necesitamos dos para comunicar cada par de habitaciones (uno para ir y otro para volver). Como todos, excepto las puertas especiales, son objetos del mismo tipo, en la base de datos no podemos observar el punto de origen y destino de cada puerta. Para tal fin se desarrolló una pequeña aplicación que listara todas las habitaciones del juego y las puertas contenidas en cada habitación junto con su origen y destino. Este aplicación se encuentra en el paquete pruebas y concretamente, en la clase ListarPuertas.

Antes de ejecutar dicha aplicación hay que ejecutar ClienteAdministrador, pues como ya es sabido es ahí donde se crea el mundo AGM.

La salida de la aplicación para el listado de las puertas es la siguiente:

```
Habitacion H1 CASA
```

```
Habitacion H10 Cafeteria
puerta:023 origen:H10 destino:H2
puerta:024 origen:H10 destino:H11
```

```
Habitacion H11 Cruce
puerta:025 origen:H11 destino:H10
puerta:026 origen:H11 destino:H16
puerta:027 origen:H11 destino:H13
```

```
Habitacion H12 exterior
puerta:040 origen:H12 destino:H15
puerta:041 origen:H12 destino:H17
puerta:042 origen:H12 destino:H23
```

Habitacion H13 paneles
puerta:028 origen:H13 destino:H11
puerta:029 origen:H13 destino:H22

Habitacion H14 Hall_fondo
puerta:030 origen:H14 destino:H2
puerta:031 origen:H14 destino:H15

Habitacion H15 Exterior1
puerta:032 origen:H15 destino:H14
puerta:033 origen:H15 destino:H16
puerta:034 origen:H15 destino:H12

Habitacion H16 Exterior2
puerta:035 origen:H16 destino:H15
puerta:036 origen:H16 destino:H17
puerta:037 origen:H16 destino:H11

Habitacion H17 Exterior3
puerta:038 origen:H17 destino:H16
puerta:039 origen:H17 destino:H12

Habitacion H18 EntradaBiblioteca
puerta:044 origen:H18 destino:H2
puerta:045 origen:H18 destino:H19

Habitacion H19 Biblioteca
puerta:046 origen:H19 destino:H18
puerta:047 origen:H19 destino:H20

Habitacion H2 HALL
puerta:01 origen:H2 destino:H3
puerta:02 origen:H2 destino:H8
puerta:03 origen:H2 destino:H6
puerta:04 origen:H2 destino:H7
puerta:05 origen:H2 destino:H10
puerta:06 origen:H2 destino:H14
puerta:07 origen:H2 destino:H18

Habitacion H20 Laboratorio
puerta:048 origen:H20 destino:H19

Habitacion H21 Balcon
puerta:049 origen:H21 destino:H4

Habitacion H22 Museo
puerta:050 origen:H22 destino:H13

Habitacion H23 Exterior5
puerta:043 origen:H23 destino:H12

Habitacion H3 BAÑO
puerta:08 origen:H3 destino:H2

Habitacion H4 PASILLO2
puerta:018 origen:H4 destino:H5
puerta:019 origen:H4 destino:H8
puerta:020 origen:H4 destino:H9
puerta:021 origen:H4 destino:H21

Habitacion H5 DESPACHO

puerta:015 origen:H5 destino:H4

Habitacion H6 CONSERJERIA
puerta:013 origen:H6 destino:H2

Habitacion H7 Secretaria
puerta:017 origen:H7 destino:H2

Habitacion H8 PASILLO1
puerta:010 origen:H8 destino:H2
puerta:011 origen:H8 destino:H4

Habitacion H9 DESPACHO2
puerta:022 origen:H9 destino:H4

Otros objetos

Hay que tener en cuenta que habrá dos tipos de objetos:

- Aquellos que se crean de inicio y que no son incorporados a inventario.
- Los que son recogidos por un jugador y por tanto, son incorporados a su inventario.

Los primeros tienen un identificador constante, que se puede consultar en la base de datos, mientras que los segundos se crean cuando el jugador los recoge y pertenecen a dicho jugador, de modo que su identificador estará formado por el nombre de la clase del objeto unido al identificador del jugador al que pertenecen.

Para trabajar con este segundo tipo de objetos en las pruebas (de los cuales no sabemos su identificador hasta que conocemos el identificador del jugador al que pertenecen, es decir hasta que se ha creado el jugador), se ha tomado como convenio que en las pruebas aparezcan con el nombre de la clase a la que pertenecen seguido de los caracteres ^{ao}. De este modo, cuando la aplicación se encuentre con dichos caracteres los sustituirá por el identificador del jugador correspondiente.

Los objetos que se encuentran en AGM son los siguientes:

O12 agm.objetos.objetosNoPersonaje.Taquilla
O14 agm.objetos.objetosNoPersonaje.Llave
O16 agm.objetos.objetosNoPersonaje.CajaDeCarnes
O9 agm.objetos.objetosNoPersonaje.Pastillero

Los objetos que pueden pertenecer a un jugador en concreto, por lo que no se puede conocer su identificador de inicio son:

agm.objetos.objetosNoPersonaje.Aprobado
agm.objetos.objetosNoPersonaje.AprobadoSOS
agm.objetos.objetosNoPersonaje.CarneI
agm.objetos.objetosNoPersonaje.CarneJ
agm.objetos.objetosNoPersonaje.Jabon
agm.objetos.objetosNoPersonaje.Llave_taquilla

Personajes no jugadores

Los siguientes personajes no jugadores son los que aparecen en el juego:

PNJ1	Conserje	agm.objetos.personajesNoJugadores.Conserje
PNJ2	Vaquero	agm.objetos.personajesNoJugadores.ProfesorX
PNJ3	Secretario	agm.objetos.personajesNoJugadores.Secretario
PNJ4	Ferretero	agm.objetos.personajesNoJugadores.Ferretero
PNJ5	ProfesorSOS	agm.objetos.personajesNoJugadores.ProfesorSOS

3.7.2. *Acciones a efectuar*

Sobre los objetos y los personajes no jugadores se pueden efectuar diversas acciones, algunas tendrán éxito (pues se ha pensado en el diseño del objeto o personaje no jugador que esa sea su función) y otras no nos servirán para nada pues en el diseño del objeto no se pensó que dicha acción sobre dicho objeto llevase a algo. Aún así, como norma, podremos efectuar cualquier acción sobre cualquier objeto o personaje no jugador, recibiendo por parte del sistema la respuesta que sea conveniente en cada caso.

Las acciones que se pueden realizar son:

SOLICITUD CREACION

Crea a un personaje jugador.

SOLICITUD CONEXIÓN

Conecta a un personaje jugador previamente creado.

IR A <objeto (normalmente una puerta)>

Nos trasladará a otra estancia.

MIRAR <objeto/personaje no jugador>

Nos mostrará una descripción del objeto o del personaje.

HABLAR <personaje no jugador> <opciones de conversación>

Iniciaremos una conversación con el personaje no jugador indicado, opcionalmente se pueden especificar las opciones elegidas para guiar el dialogo.

COGER <objeto>

Añadirá al inventario el objeto, si es posible.

USAR <objeto/personaje no jugador> <objeto/personaje no jugador>

Aplicará un parámetro sobre el otro, si es posible, de manera que obtendremos ciertos resultados.

3.7.3. *Escribir la prueba*

Una vez tenidos en cuenta los puntos anteriores ya podemos escribir el conjunto de acciones que queramos realizar sobre los objetos y/o personajes no jugadores. Solo tendremos que decidir cuantos usuarios queremos que realicen tales acciones y si queremos que esos jugadores realicen las mismas acciones o distintas. Así que simplemente tendremos que estructurar nuestra prueba siguiendo la plantilla ya expuesta en el apartado *Plantilla para pruebas*.

3.8. Aplicación a un caso práctico: 10% de azar

En este apartado se describe como se ha realizado el documento XML que codifica las acciones necesarias para llevar a buen fin la aventura 10% de azar.

3.8.1. Puertas implicadas

En esta aventura se han de atravesar múltiples puertas:

- Del hall al pasillo1: O2
- Del pasillo1 al pasillo2: O11
- Del pasillo2 al despacho: O20
- Del despacho al pasillo2: O22
- Del pasillo2 al pasillo1: O19
- Del pasillo1 al hall: O10
- Del hall a secretaria: O4
- De secretaría al hall: 017

3.8.2. Objetos implicados

Los objetos con los que hay que interactuar son los siguientes, junto con las acciones a efectuar sobre ellos:

- Usar la llave de la taquilla (Llave_taquilla^{ao}) con la taquilla (O12)
- Coger el carneJ (carneJ^{ao}) de la taquilla (coger taquilla)
- Coger un carne de la caja de carnés (O16)
- Usar el carneJ (carneJ^{ao}) y el carneI (carneI^{ao})

3.8.3. Personajes no jugadores implicados

Sólo hay que tener en cuenta a dos personajes no jugadores: el profesorSOS (PNJ5) y el secretario (PNJ3). Sobre los que hay que efectuar acciones de hablar y usar.

3.8.4. Acciones involucradas

Ya hemos visto que acciones hay que efectuar con cada objeto o personaje no jugador. Ahora vamos a detallarlo, explicando el por qué de cada acción:

- SOLICITUD CREACION: para crear al jugador
- SOLICITUD CONEXION: para conectar al jugador
- IR A O2: para ir al pasillo1
- IR A O11: para ir al pasillo2
- IR A O20: para ir al despacho2
- HABLAR PNJ5: para hablar con el profesorSOS
- IR A O22: para ir al pasillo2
- IR A O19: para ir al pasillo1
- USAR Llave_taquilla^{ao} O12: Para abrir la taquilla con la llave
- COGER O12: para coger nuestro carne de la taquilla
- IR A O11: para ir al pasillo2
- IR A O20: para ir al despacho2
- HABLAR PNJ5: para hablar con el profesorSOS
- USAR PNJ5 CarneJ^{ao}: para darle nuestro carné al profesorSOS
- IR A O22: para ir al pasillo2

- IR A O19: para ir al pasillo1
- IR A O10: para ir al hall
- IR A O4: para ir a secretaria
- HABLAR PNJ3 1S: para hablar con el secretario y que se vaya
- COGER O16: para coger un carné de la caja
- IR A O17: para ir al hall
- IR A O2: para ir al pasillo1
- IR A O11: para ir al pasillo2
- IR A O20: para ir al despacho2
- HABLAR PNJ5: para hablar con el profesorSOS
- USAR PNJ5 CarneI^{ao}: para darle el otro carné al profesorSOS

3.8.5. *Escribir la prueba de la aventura*

Sólo tendremos un jugador involucrado que hará las acciones explicadas anteriormente, de modo que la codificación de la prueba será la siguiente:

```
<?xml version="1.0"?>

<pruebas xmlns="pruebas.xsd">
  <prueba nombre="PruebaCreacionPJ">
    <numPJs>1</numPJs>
    <acciones>
      <accion tipo="SOLICITUD CREACION"/>
      <accion tipo="SOLICITUD CONEXION"/>
      <accion tipo="IR A" parametro="O2"/>
      <accion tipo="IR A" parametro="O11"/>
      <accion tipo="IR A" parametro="O20"/>
      <accion tipo="HABLAR" parametro="PNJ5"/>
      <accion tipo="IR A" parametro="O22"/>
      <accion tipo="IR A" parametro="O19"/>
      <accion tipo="USAR" parametro="Llave_taquillaao" opciones="O12"/>
      <accion tipo="COGER" parametro="O12"/>
      <accion tipo="IR A" parametro="O11"/>
      <accion tipo="IR A" parametro="O20"/>
      <accion tipo="HABLAR" parametro="PNJ5"/>
      <accion tipo="USAR" parametro="PNJ5" opciones="CarneJao" />
      <accion tipo="IR A" parametro="O22"/>
      <accion tipo="IR A" parametro="O19"/>
      <accion tipo="IR A" parametro="O10"/>
      <accion tipo="IR A" parametro="O4"/>
      <accion tipo="HABLAR" parametro="PNJ3" opciones="1S"/>
      <accion tipo="COGER" parametro="O16"/>
      <accion tipo="IR A" parametro="O17"/>
      <accion tipo="IR A" parametro="O2"/>
      <accion tipo="IR A" parametro="O11"/>
      <accion tipo="IR A" parametro="O20"/>
      <accion tipo="HABLAR" parametro="PNJ5"/>
      <accion tipo="USAR" parametro="PNJ5" opciones="CarneIao" />
    </acciones>
  </prueba>
</pruebas>
```

3.9. *Ejecutando la aplicación*

La interfaz de la aplicación es simple. Sólo consta de un JOptionPane en el que se puede elegir mediante menú desplegable entre los documentos XML que se encuentren en el directorio de trabajo actual.

Una vez seleccionada la prueba que queramos que el sistema lleve a cabo no queda más que pulsar sobre el botón aceptar con lo que si todo es correcto la prueba elegida será ejecutada sobre AGM y podremos seguir su evolución por pantalla.

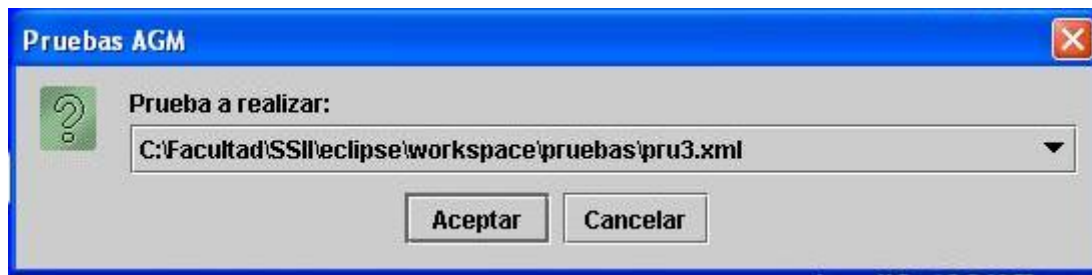


Ilustración 9. Interfaz de la aplicación

En la consola del sistema podremos ir viendo como evoluciona la prueba, si se van realizando correctamente todas las acciones especificadas en el documento XML, si hay algún fallo, etc.

Cuando se conecten los personajes jugadores implicados en la prueba seleccionada se abrirá una ventana por jugador. De modo que podremos seguir las evoluciones de todos los jugadores, cada uno en su pantalla; y si procede, veremos como se encuentran y evolucionan conjuntamente.

3.10. Integración con AGM

El paquete pruebasxml es un paquete aparte del juego pero que por su funcionalidad, debe hacer uso del mismo mediante llamadas a métodos de las clases ClienteAdministrador y ClienteJugador. Es por este motivo, que aunque en principio cualquier cambio en la estructura de AGM no debiera afectar al correcto funcionamiento del paquete, hay que cumplir una serie de normas con respecto a las clases anteriormente mencionadas y en concreto a los métodos de las mismas que se emplean en el paquete de pruebas.

3.10.1. ClienteAdministrador

Con respecto a esta clase son pocas las normas que hay que cumplir puesto que el uso que se hace dentro del módulo de pruebas se limita a hacer llamadas a los métodos borraMundo() y creaMundo() y también, como no, al constructor de la clase, ClienteAdministrador().

Es por esta razón que todos esos métodos han de ser de acceso público para otras clases aunque no correspondan al mismo paquete.

Estas llamadas se efectúan desde la clase PruebasParser, y en concreto, dentro del main() de la aplicación.

3.10.2. ClienteJugador

Con respecto a esta clase el asunto se complica un poco más, pues son múltiples las llamadas que se efectúan sobre objetos de este tipo. Además de existir el problema de que se debe permitir una asociación navegada entre las clases ClienteJugador y PruebaCliente, de manera que se pueda seguir la ejecución de una a otra y viceversa desde distintos puntos.

A continuación enumeramos las premisas que se han de seguir en ClienteJugador para que no se viole el correcto funcionamiento de la aplicación:

- Ha de existir un atributo privado de tipo PruebaCliente.
- Ha de existir un método público solicitudCreacionPJ() que reciba como atributo un objeto de clase DatosCreacion. De modo, que tenga la siguiente cabecera:

```
public void solicitudCreacionPJ(DatosCreacion d)
```

- Ha de existir un método público solicitudConexionPJ() que reciba como atributo un objeto de clase DatosConexion. De modo, que tenga la siguiente cabecera:

```
public void solicitudConexionPJ(DatosConexion d)
```

- Ha de existir un constructor con un parámetro para la clase, cuyo parámetro sea un objeto de tipo PruebaCliente. Este constructor hará las mismas acciones que el constructor sin parámetros de la clase, pero además, asignará el parámetro pasado al constructor al atributo antes mencionado de tipo PruebaCliente. Este constructor tiene la siguiente implementación:

```
public ClienteJugador(PruebaCliente pc){
    try{
        ctx = getContext();
        qC = creaQueueConnection();
        tC = creaTopicConnection();
        colaKernel = (javax.jms.Queue)ctx.lookup("queue/COLA_KERNEL");
        estadoFrase = 0;
        this.pc = pc;
    }
    catch(Exception e){}
}
```

- Hay que efectuar un pequeño cambio en el método onMessage(), de forma que si no hemos introducido ningún valor al atributo de tipo PruebaCliente siga el funcionamiento que tenía (esto ocurrirá si estamos haciendo llamadas desde AGM directamente, no desde el paquete de pruebas); mientras que si el atributo tiene algún valor (es decir, estamos llamando a este método desde el módulo de pruebas) retornemos al método onMessage() pero de PruebaCliente. Por tanto, este método tendrá que tener o respetar la siguiente forma:

```
public void onMessage(Message message){
    try{
        if(pc==null)
            procesa(message);
        else
        {
            System.err.println("onmessage no null");
            pc.onMessage(message);
        }
    }
    catch(Exception e){
        e.printStackTrace();
    }
}
```

- Hay que sacar del método procesa(javax.jms.Message message) todo lo relativo a las distintas acciones a hacer dependiendo del tipo de mensaje recibido e introducirlo en nuevos métodos públicos. De modo, que tendremos las siguientes cabeceras:

```
public void muestraPantalla(javax.jms.Message message)
public void refresco(javax.jms.Message message)
public void notificacion(javax.jms.Message message)
public void comunicacion(javax.jms.Message message)
```

3.11. Mensajes del servidor

Este punto trata de echar luz para comprender que tipo de mensajes llegarán por parte del servidor cuando se manda ejecutar una acción u otra y esta ha sido realizada con éxito.

ACCIÓN	TIPOS DE MENSAJE
SOLICITUD CREACIÓN	CONFIRMACION_CREACION_PJ
SOLICITUD CONEXIÓN	CONFIRMACION_CONEXION_PJ
IR A	NOTIFICACION_TEXTUAL NOTIFICACION_TEXTUAL OBJETOS_REFRESCO
COGER	NOTIFICACION_TEXTUAL
HABLAR	OBJETO_COMUNICACION
MIRAR	NOTIFICACION_TEXTUAL
USAR	NOTIFICACION_TEXTUAL

La importancia de estos mensajes para nuestra aplicación radica en el hecho de que nosotros no ejecutaremos una acción hasta que tengamos confirmación de que se ha ejecutado correctamente la anterior, es decir, hasta que nos ha llegado el mensaje o mensajes correspondientes por parte del servidor.

3.12. Probando AGM

A continuación se exponen las pruebas (documentos XML) llevadas a cabo junto a una breve descripción de las mismas.

FICHERO DE PRUEBA	DESCRIPCIÓN
pruebaCreacionPJ.xml	Creación y conexión de un personaje jugador.
pru2.xml	Creación, conexión de un personaje jugador y traslado del mismo al baño, usando la puerta O1.
pru3.xml	Creación, conexión de un personaje jugador, traslado del mismo al baño, usando la puerta O1 y coger una pastilla de jabón (objeto O9).
pru4.xml	Creación, conexión de un personaje jugador, traslado del mismo al baño, usando la puerta O1, coger una pastilla de jabón (objeto O9) y retorno al hall usando la puerta O8.
pru5.xml	Creación, conexión de un personaje jugador, traslado del mismo al baño, usando la puerta O1 y retorno al hall usando la puerta O8.
pru6.xml	Creación, conexión de un personaje jugador, traslado del mismo al baño, usando la puerta O1, retorno al hall usando la puerta O8 y traslado a la conserjería usando la puerta O4.
pru7.xml	Creación, conexión de un personaje jugador, traslado del mismo a la

	conserjería usando la puerta O4 y retorno al hall usando la puerta O17.
pru8.xml	Creación, conexión de un personaje jugador y mantenimiento de una conversación con el personaje no jugador Conserje (PNJ1) con opciones 3A y 3B.
pru9.xml	Creación, conexión de un personaje jugador, traslado del mismo al baño, usando la puerta O1 y coger dos pastillas de jabón (objeto O9).
pru10.xml	Creación, conexión de un personaje jugador, traslado del mismo al baño, usando la puerta O1, mirar el pastillero (objeto O9) y coger una pastilla de jabón (objeto O9).
pru11.xml	Creación, conexión de un personaje jugador, traslado del mismo al baño, usando la puerta O1, mirar el pastillero (objeto O9) y retorno al hall usando la puerta O8.
pru12.xml	Creación, conexión de un personaje jugador, traslado del mismo al baño, usando la puerta O1, mirar el pastillero (objeto O9), retorno al hall usando la puerta O8 y traslado a la conserjería usando la puerta O4.
pru13.xml	Creación, conexión de un personaje jugador, traslado del mismo al baño, usando la puerta O1, mirar el pastillero (objeto O9), coger una pastilla de jabón (objeto O9) y retorno al hall usando la puerta O8.
pru14.xml	Creación, conexión de dos personajes jugadores, traslado de los mismos al baño, usando la puerta O1, mirar el pastillero (objeto O9), coger una pastilla de jabón (objeto O9) y retorno al hall usando la puerta O8.
otra.xml	Formada por dos pruebas distintas sobre dos personajes jugadores distintos. En la primera de ellas el personaje jugador es creado, conectado, va al baño (puerta O1), coge una pastilla de jabón (objeto O9) y vuelve al hall (puerta O8). En la segunda, el otro personaje jugador es creado, conectado y realiza la acción de hablar con el personaje no jugador Conserje (PNJ1).
pru15.xml	Creación, conexión de un personaje jugador, traslado del mismo al baño, usando la puerta O1, mirar el pastillero (objeto O9), coger una pastilla de jabón (objeto O9), retorno al hall usando la puerta O8, ir a secretaría (puerta O4),

	hablar con el secretario con opciones 1S y coger un carné de la caja de carnés (O16).
pru_monitorizacion.xml	Creación, conexión de tres personajes jugadores, cada uno de los cuales se envía a una habitación distinta: el baño (puerta O1), secretaría (puerta O4) y el pasillo1 (puerta O2).
Prueba_ElsecretoDelProfesor.xml	Creación, conexión de dos personajes jugadores. El primero se pondrá a hablar con el conserje (PNJ1), mientras el segundo irá al baño, cogerá una pastilla de jabón, volverá al hall para entrar en conserjería y conseguir la llave de un despacho. Usará la llave con el jabón para conseguir un molde. Saldrá de conserjería e irá al pasillo2 donde hablará con el ferretero (PNJ4) para que le haga una copia de la llave. Con dicha llave abrirá la puerta del despacho y hablará con Vaquero (PNJ2).
10%_de_azar.xml	Prueba completa de la aventura 10% de azar.
10%_de_azar_min.xml	Prueba simplificada de la aventura 10% de azar.
10%_de_azar_peq.xml	Prueba con el mínimo de acciones posibles para probar la aventura 10% de azar.
pru_acciones.xml	Creación, conexión de un personaje jugador, traslado del mismo al pasillo1 (puerta O2), uso de la llave de la taquilla (Llave_taquilla ^{ao}) con la taquilla (O12), coger el carneJ (carneJ ^{ao}), ir al pasillo2 (puerta O11), ir al despacho2 (puerta O20), hablar con el profesorSOS (PNJ5) y usar con él el carnéJ.
varios.xml	Creación y conexión de 10 personajes jugadores y traslado de todos ellos al baño. De manera que probemos la posibilidad de la conexión múltiple al AGM.

La metodología seguida para ir llevando a cabo unas pruebas u otras fue la de ir probando de forma incremental nuestra aplicación, empezando por pruebas muy básicas, como la de la creación y conexión de un personaje jugador hasta llegar a pruebas complejas como la de 10%_de_azar.xml. Hay algunas pruebas que tienen como fin probar alguna funcionalidad concreta del sistema como la prueba varios.xml que trata de mostrar la posibilidad del sistema de soportar múltiples conexiones de distintos usuarios sin que el sistema se resienta.

3.13. Conclusiones

A partir del módulo aquí expuesto y desarrollado se puede seguir progresando en esta área y mejorar funcionalidades, como por ejemplo, alcanzar el sincronismo entre distintos personajes jugadores, de modo que se esperen mutuamente con el fin de lograr un objetivo común.

Resulta de extrema importancia el campo de las pruebas y verificación en sistemas de alta complejidad y funcionalidad crítica, que aunque el caso de AGM no es de tales características, esta experiencia resulta enriquecedora de cara a afrontar retos en otras áreas que sí tienen estas propiedades.

También hay que reseñar que un juego no probado de forma conveniente nunca puede ser lanzado al mercado, si no se quiere conseguir mala publicidad y que no se nos tome en serio en un futuro. De modo, que es importante el testeo en un producto que va a ser chequeado de forma concisa por los usuarios finales, los cuales cada vez más tienen un mayor nivel de exigencias.

4. Animación en la AGM

4.1. Introducción

Con el objetivo de mejorar la apariencia visual del juego, se ha buscado la forma de animar a los personajes presentes en la habitación actual. Para esto, se ha llevado a cabo un estudio consistente en la elaboración de un conjunto de pruebas de investigación con una librería de código abierto encontrada en internet y JavaSwing. Adicionalmente, aunque en una fase muy temprana –experimental, podríamos decir, de hecho – se ha comenzado la aplicación e integración del resultado de las pruebas a la aplicación global.

Para implementar las animaciones seguiremos una aproximación ya clásica en el mundo de los videojuegos: animación basada en sprites bidimensionales. Un sprite no es más que un mapa de bits, generalmente de pequeño tamaño y parcialmente transparente (para no limitarnos a imágenes rectangulares), con el que representaremos a los personajes, objetos, etc, etc. El efecto de movimiento se consigue modificando la imagen que se muestra en la pantalla, sustituyéndola por otra. Sirva de ejemplo la siguiente figura:



Ilustración 10: Ejemplo de sprites que conforman una animación

El conjunto de imágenes representado arriba correspondería en realidad a un único movimiento. Si vamos sustituyendo con una frecuencia suficientemente alta la imagen mostrada en pantalla por la posterior, el ojo humano tendría la ilusión de estar viendo al mono moviéndose.

A continuación, tras detallar en el siguiente subapartado los requisitos buscados para la animación, pasaremos a hablar de las propias pruebas realizadas para comprobar el funcionamiento de la librería y extender sus capacidades de acuerdo con las posibles necesidades y funcionalidades requeridas para nuestra propia aplicación. Tras una exposición de la evolución de los conceptos “teóricos” que se implementarían, o al menos intentaría implementar, se pasará a explicar la estructura de clases, métodos y funcionalidad de los mismos, aplicabilidad de estos a la hora de añadir finalmente la animación de los objetos a la aventura gráfica,....

Para evitar tener que comenzar desde cero se buscó por Internet una librería de código abierto que hiciera posible reutilizar el código Java correspondiente a la definición de estructuras de datos y operaciones relacionadas con la gestión básica de sprites y animaciones. Comentaremos brevemente en el apartado posterior la librería encontrada (y utilizada), *gamelib*.

Posteriormente, dedicaremos también un apartado a la estructura de los ficheros XML que almacenarán toda la información necesaria para configurar las animaciones: la lista de posibles movimientos/estados, las imágenes necesarias para cada movimiento,...

Por último, se expondrán una serie de directrices encaminadas a la integración final de los conceptos desarrollados durante las pruebas con la *Aventura Gráfica Multijugador*.

4.1.1. Requisitos

La inclusión de animación en la aplicación debería satisfacer una serie de restricciones:

- Usuarios en una misma habitación deben ver los mismos objetos y personajes en las mismas posiciones. (Nota: de ahora en adelante, utilizaremos objetos para referirnos indistintamente a objetos o personajes)
- Las animaciones de un objeto/personaje han de poder cambiarse, en respuesta a acciones desencadenadas por el usuario, interacciones con otros objetos, o respuesta “automática” al darse ciertas condiciones en el entorno. Tales variaciones, de acuerdo con el primer requisito, deberán ser comunicadas a los otros usuarios.
- Algunos objetos móviles deben tener limitado el ámbito de movimiento a una región poligonal.

4.2. Pruebas de investigación.

La idea inicial de las pruebas era comprobar las capacidades de la librería, y su aplicabilidad a nuestro proyecto, extendiendo la funcionalidad de la misma. Durante esta fase, se desarrolló una serie de clases centradas en una nueva capacidad, y cómo podría resultar útil a la hora de efectuar la integración con la aplicación real.

Así, en el primer apartado del presente epígrafe, se expondrán dichas ideas conceptualmente, sin entrar en detalles de implementación, y la evolución de las mismas con el tiempo. La mayoría de dichas ideas fueron concretándose en los foros de discusión destinados a la comunicación entre los miembros del proyecto.

Posteriormente se mostrará la estructura de clases del programa de pruebas con un mayor detenimiento del que tuvimos explicando los principales paquetes de la librería gamelib, deteniéndonos a explicar en profundidad el funcionamiento de los métodos nuevos más importantes.

Finalmente se expondrán las conclusiones a que se llegó durante la fase de experimentación.

4.2.1. Ideas teóricas. Evolución de las mismas.

Impresiones iniciales

Uno de los primeros puntos que se discutió fue la animación del personaje jugador (PJ). Además de en la propia pantalla, es necesario que tenga lugar en la pantalla de todos

aquellos usuarios conectados que se encuentren en la misma habitación que nuestro personaje. Así, el programa debería soportar múltiples animaciones; tanto aquellas ordenadas desde el cliente – que afectan al propio personaje, así como las animaciones de personajes no jugadores (PNJs), y objetos no personajes (ONPs) - como las ordenadas desde el servidor – principalmente, personajes de otros.

La primera posibilidad consistiría en enviar mensajes desde el servidor cada vez que fuera necesario realizar un movimiento, lo que introduciría una elevada carga al servidor, pudiéndolo saturar con un número relativamente bajo de usuarios conectados. Más adecuada, sin embargo, resultaba la otra posibilidad: Definir un protocolo de “intenciones”, que serían enviadas por el servidor, y gestionadas por los clientes. Por poner un ejemplo, si un usuario quisiera moverse de un punto P1 a otro P2, en vez de ir enviando una secuencia de movimientos individuales que cubrieran la trayectoria P1-P2, el servidor enviaría un mensaje tal como “QUIERO DESPLAZARME DE P1 A P2”. Con esto sería también posible abortar el movimiento, en caso, por ejemplo, de un redireccionamiento, si el usuario quisiera moverse a otro lugar.

Otro aspecto importante es el referido a no limitar la animación exclusivamente a PJs, pudiendo dar a los PNJs, o incluso a los ONPs, la capacidad de moverse, y en general, de estar animados. Inicialmente se pensó en que tuvieran una única secuencia de movimientos asociada, que se ejecutara constantemente (por ejemplo, dar vueltas alrededor de un área, permanecer quietos respirando,... esto queda a la imaginación de cada uno) , e incluso una opción más “de grano grueso”, consistente en emplear archivos GIF animados, para animaciones sin movimiento, que también parecía funcionar bien. La decisión final en relación a esto consistió en tener un conjunto de movimientos asociados a una serie de estados, lo que resultaría plenamente compatible con la animación de los PJs, en el sentido de que podría definirse la animación a nivel de objeto, y no sólo de personajes. Así, no estaríamos limitados a una única animación que se repetiría ad infinitum, pudiendo variar, o incluso responder ante acciones de los usuarios (por ejemplo, que un PNJ, al hablar con un personaje, cambiara su animación “por defecto” a “hablando”, para luego volver a la anterior, o incluso a otra nueva).

Configuración de los parámetros para las animaciones

La decisión con que se llegó al final de la subsección precedente condujo a otra igualmente importante: la utilización de ficheros de configuración para las animaciones, especificando el tipo de animación a emplear, los gráficos necesarios, ...Veremos más adelante cómo se realizó esto, y el formato de los mismos.

Primer prototipo para la aplicación de testeo de los resultados de la investigación

Una vez establecido el método básico de animación/movimiento (basándonos en secuencias de movimientos, o acciones, asociadas a un estado) dieron comienzo las primeras pruebas, consistentes en el desarrollo de una aplicación gráfica mínima sobre la que se añadirían distintos sprites con animación y/o movimiento.

Posteriormente, sobre el desarrollo de las pruebas básicas iban surgiendo otras dudas, que darían lugar a nuevas ideas. Así, una de las primeras fue la posibilidad de cambiar la secuencia de movimientos actual en respuesta a determinados eventos, bien de la

interfaz de usuario (por ejemplo, un clic de ratón), o bien del “entorno”. En este segundo grupo destacaba especialmente la respuesta ante la colisión contra los bordes del área permitida sobre la cual el sprite puede desplazarse, pues motivó además otro objetivo adicional.

En las primeras pruebas, el único sprite existente era una araña que se movía por el suelo a derecha o izquierda. Esto se representaba indicando en el estado de la araña que se estaba moviendo en esa dirección concreta. Al llegar a los bordes de la pantalla, en lugar de detenerse (respuesta por defecto), se pretendía que cambiara el estado (y con él, toda la secuencia de movimientos), a “me estoy moviendo en la dirección contraria”. Si esto era posible, de igual modo sería factible un cambio de estado como respuesta a cambios en el contexto del sprite. El resultado fue satisfactorio.

Por el contrario, el segundo objetivo que surgió no fue tan fructífero (aunque la idea es buena): las animaciones para un movimiento se configuran con una lista de imágenes. En el caso de movimientos simétricos (por ejemplo, moverse a la izquierda o a la derecha) no es diferente: se utiliza una lista para el movimiento 1, y otra para el movimiento 2. Pues bien, para evitar utilizar una de las listas, con la consiguiente mejora en eficiencia al ahorrar la carga de las imágenes, se pensó en implementar uno de los movimientos a partir de las imágenes de otro, aplicando una matriz de transformación (típicamente, reflejar horizontal o verticalmente la imagen). Esta idea, si n embargo, no llegó a buen puerto y se acabó desechando.

Respuesta ante eventos, y cambios de animación asociados a cambios en el estado del objeto

En pruebas posteriores, siguiendo con un único sprite, se consiguió definir una animación “temporal”, que pasara a otra diferente al finalizar, de forma automática. La utilidad de esto radica en que permite la descomposición de una animación compleja en varias animaciones (no individuales). Por ejemplo, así podríamos implementar la animación para una acción de tipo “coger un objeto alejado del personaje” como sigue: en primer lugar, tendríamos una animación de desplazamiento hasta llegar al objeto, y una vez que llega a su objetivo(o bien detecta la colisión contra ese mismo objeto), pasar a ejecutar la animación propia de coger el objeto.

Además de esto, se consiguió la respuesta ante eventos de ratón (en la aplicación de prueba esto permitía indicarle a la araña un punto al que debía moverse, en lugar de estar dando vueltas a un área fija), menos simples en principio que los clásicos eventos de pulsación de botones. La utilidad más inmediata, como cabría esperar, se encuentra en la posibilidad de representar gráficamente la ejecución de acciones “ir a”

Movimiento dirigido

Más tarde se añadió un segundo sprite que acompañase al de la araña, en el esqueleto creado para estas pruebas. Su principal característica es que podía desplazarse de un sitio a su objetivo, especificado tras hacer clic con el ratón sobre la pantalla, utilizando un vector como guía para el movimiento. Esto nos permitía disponer de distintas clases de animación, y no limitarnos a utilizar únicamente secuencias (mejor dicho, patrones) de movimientos, que para algunas acciones sencillas tal vez resultaran excesivas. Aunque se tuvo éxito en estas pruebas, finalmente se ha optado por utilizar en todos los casos sprites animados utilizando un patrón de movimientos. Un tercer sprite fue

agregado para probar la misma funcionalidad que el segundo, pero animado íntegramente con un patrón.

Área permitida de movimiento y detección de bordes

Las últimas pruebas estaban relacionadas con la detección de colisiones contra otros sprites, y contra los bordes del área de movimiento. Para el primer problema, existían las siguientes opciones:

- Detenerse al detectar un obstáculo (el otro sprite). Correspondería al usuario buscar otra ruta libre de colisiones.
- Imbuir al sprite de capacidad para bordear el obstáculo, y adaptar su trayectoria.
- Permitir solapamiento de sprites, es decir, mantener la opción por defecto.

Por falta de tiempo, se optó por la última opción, mucho más sencilla que la primera (que, sin embargo, no reviste una especial complicación, es un caso particular de respuesta ante eventos del entorno que mencionamos antes), e infinitamente más fácil de implementar que la segunda, que sí implicaba ya una cierta complejidad.

Sobre el segundo problema, la librería proporciona ya una solución. Define un área rectangular, que caracteriza la región por la que el sprite tiene libertad de movimiento. Al impactar contra los bordes del área, existe un cierto número de respuestas predeterminadas: no hacer nada, detenerse, rebotar contra el borde, o aparecer por el borde contrario. Adicionalmente, demostramos en líneas precedentes que era posible ampliar el número de comportamientos posibles.

El mayor inconveniente de la solución “por defecto” radica en que nos limitamos a áreas rectangulares, lo que resulta poco flexible.

Llegados a este punto, se pensaron dos soluciones adicionales, más flexibles. La primera de ellas consistía en definir por medio de un conjunto de rectángulos el área de movimiento, y poder simular áreas de distintas formas por aproximación. La segunda de ellas sustituía el área rectangular por una poligonal. En este caso se perseguía aplicar conceptos impartidos en la asignatura de Informática Gráfica. Sin embargo, se desechó la idea por su complejidad relativa. La primera de las soluciones propuestas parte (o mejor dicho, partía, pues se le dio una vuelta de tuerca posteriormente) de la idea que se muestra en la figura:

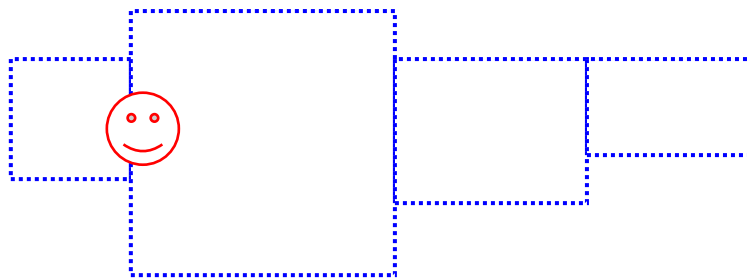


Ilustración 11: Establecimiento de la región de movimiento por medio de rectángulos

El mecanismo de detección consistía básicamente en recorrer la lista de rectángulos y comprobar si el sprite excedía los límites de los rectángulos individuales. Esto, no obstante, no resultaba correcto, puesto que en gran parte de los casos el sprite podría “ocupar” varios rectángulos. Esto daba lugar a una explosión de posibilidades a tener en cuenta. A la solución de este problema se llegó dándole el enfoque complementario a la

definición de la región. En lugar de que los rectángulos definieran el área permitida, debían modelar el área prohibida. Así, la intersección entre el sprite y alguno de estos rectángulos será indicativo de que está excediendo los límites impuestos. Con esta aproximación se simplificó enormemente el algoritmo de detección de colisión contra los bordes.

Compárese la figura anterior con la siguiente:

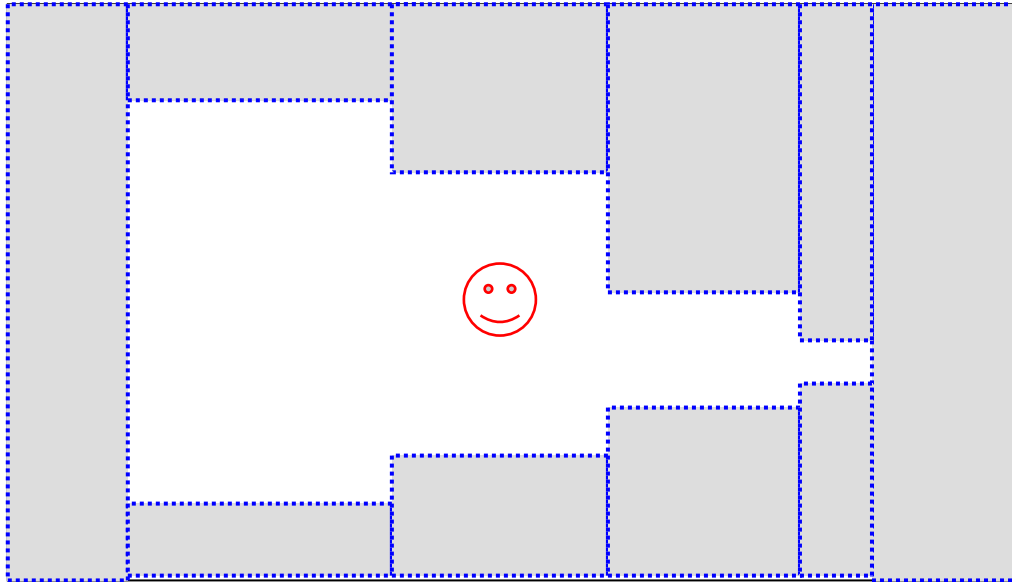


Ilustración 12: El área permitida en este caso es la región no sombreada

Y, ya por último, una captura del programa de pruebas donde se marcan los rectángulos que delimitarán el área para el sprite que aparece:

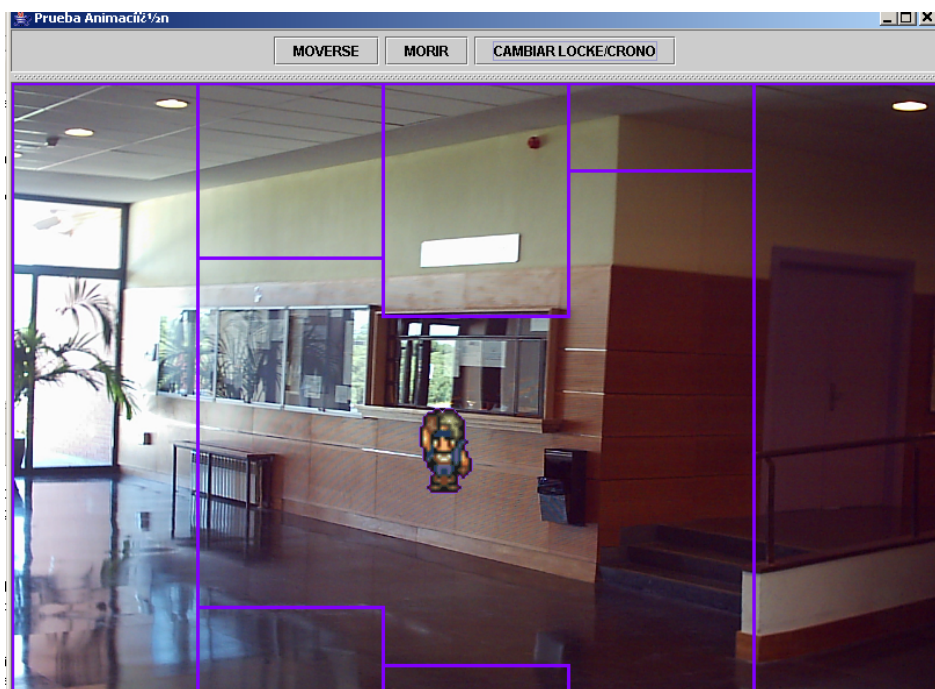


Ilustración 13: Screenshot con el aspecto del programa de pruebas delimitando los rectángulos

El desarrollo práctico de todas estas ideas se fue fraguando en el programa para pruebas, que constituyó la base sobre la cual se han establecido las funcionalidades que deberían poseer los sprites que nos pudieran resultar de utilidad en la aplicación final. La estructura del mismo se verá en el siguiente apartado.

4.2.2. Estructura del programa de pruebas

Para las pruebas necesarias con el fin de desarrollar las ideas teóricas comentadas en el subapartado anterior, se implementó una sencilla aplicación, basada en los programas de ejemplo proporcionados con la librería *gamelib*, sobre la cual se añadirían los sprites prototipo. A continuación, se muestran los siguientes diagramas de clases, correspondientes a la aplicación de prueba.

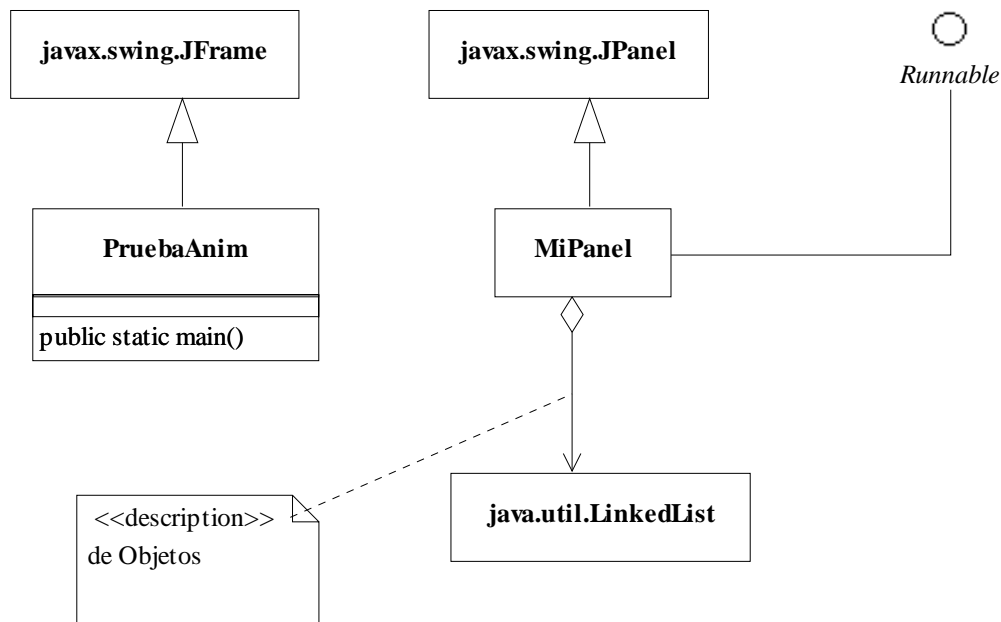


Ilustración 14: Estructura de la aplicación, parte (a)

En la figura a) Se muestran, de forma simplificada, las clases principales involucradas en el diseño de la aplicación, sin tener en cuenta por el momento la jerarquía de sprites. Así, sin entrar todavía en detalles, el método `main()` que inicia el flujo de ejecución, se encuentra en la clase `PruebaAnim`. Ésta representa la interfaz gráfica, constituida por un `Jframe` que contiene una botonera, y la pantalla, que no es más que una instancia de `MiPanel`. Los distintos sprites que se visualizarían en la pantalla se encuentran almacenados en una lista polimórfica. La jerarquía de sprites se muestra en el subdiagrama siguiente (en realidad, las figuras “a” y “b” corresponden a un mismo diagrama, pero por visibilidad se han separado en dos).

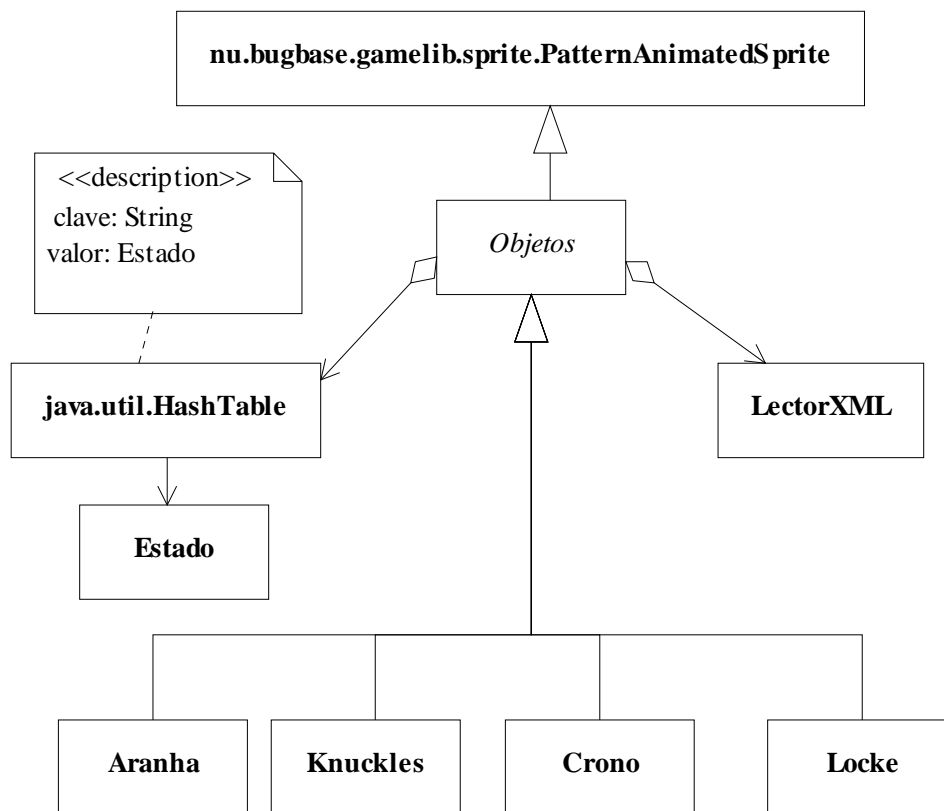


Ilustración 15: Jerarquía de clases para representar los distintos objetos “Sprite”. Estructura de la aplicación, parte (b)

En la figura “b” se representa, ahora sí, la jerarquía de clases para los sprites que se han utilizado. Se muestra a la cabeza de dicha jerarquía la clase `PatternAnimatedSprite`, perteneciente a `gamelib`, pues aquí es donde enlazamos con la jerarquía de Sprites de la propia librería. La clase abstracta `Objetos` proporciona una base común a nuestros sprites. Como vemos, posee un `LectorXML`, clase desarrollada para el procesamiento de los ficheros de configuración, y una tabla hash de estados. Dichos estados, que se inicializan al leer el XML de configuración, constan de un patrón de movimiento (una instancia de la clase `MovePattern`), el nombre y un tipo, que finalmente no se utiliza, que determinaría el tipo de animación (guiada por vector, por patrón,...) a emplear para ese estado concreto.

Interfaz Gráfica

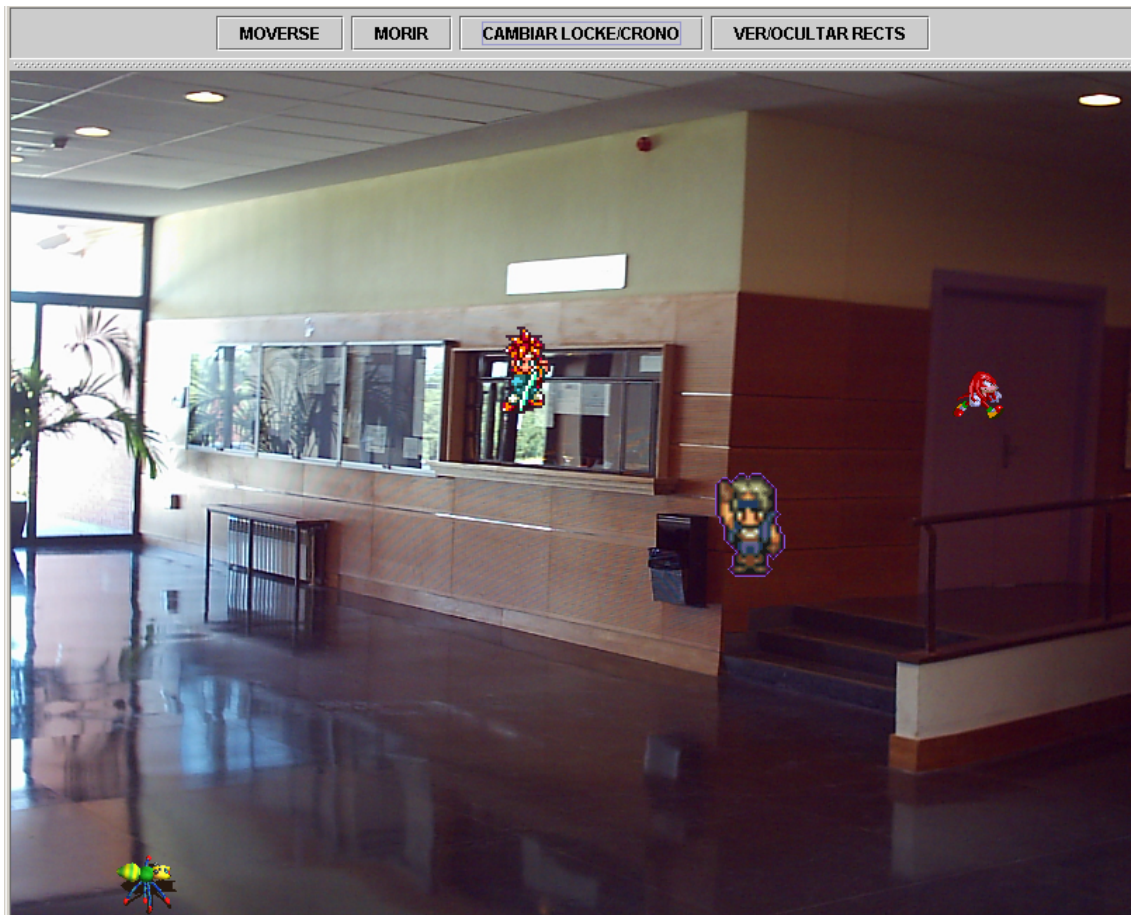


Ilustración 16: Captura de pantalla correspondiente a la interfaz al completo

En la imagen aparece un screenshot de la interfaz gráfica (la barra de títulos se ha recortado por visibilidad), en la que puede verse tanto la botonera como la pantalla principal, así como los sprites moviéndose por ella. Como puede apreciarse fácilmente, la interfaz tiene un diseño extremadamente simple.

PruebaAnim.java

Esta clase representa la interfaz gráfica completa. Además, es la que posee el método `main()`, de entrada al programa. Sus métodos inicializan los distintos elementos de la interfaz.

- **Declaracion:**

```
public class PruebaAnim extends JFrame
```

- **Importaciones:**

```
import javax.swing.JSplitPane;  
import javax.swing.JPanel;
```

```
import javax.swing.JButton;
import javax.swing.JFrame;

import java.awt.*;
import java.awt.event.*;
```

- **Atributos:**

Nombre	Tipo	<i>Descripción:</i>
V	PruebaAnim	Instancia estática de la interfaz
P	MiPanel	Instancia del panel que constituirá la “pantalla”

- **Métodos:**

<u>Nombre:</u> PruebaAnimacion()	<u>Tipo devolución:</u> No aplicable
<i>Argumentos:</i> No tiene	
<i>Descripción</i> Constructor Inicializa el JFrame.	

<u>Nombre:</u> ProcessEvent()	<u>Tipo devolución:</u> void
<i>Argumentos:</i> <ul style="list-style-type: none"> • AWTEvent evt: Evento de ventana 	
<i>Descripción:</i> Procesa eventos de ventana. Si no se trata del evento de cierre de la ventana, delega en el método de la superclase (Jframe). En caso contrario, finaliza el proceso, y con ello la aplicación	

<u>Nombre:</u> main()	<u>Tipo devolución:</u> void
<i>Argumentos:</i> <ul style="list-style-type: none"> • String[] args: argumentos (No utilizados) 	
<i>Descripción:</i> Programa principal. Finaliza la inicialización de la interfaz gráfica, muestra la ventana y arranca el motor gráfico de la pantalla.	

<u>Nombre:</u> creaBotonera()	<u>Tipo devolución:</u> Jpanel
<i>Argumentos:</i> No tiene	
<i>Descripción:</i> Crea un panel con los botones necesarios para la aplicación. Adicionalmente, añade a estos los oyentes precisos para la interacción con el usuario y la	

comunicación con el motor.

MiPanel.java

Esta clase representa la pantalla de la aplicación, y además, el motor gráfico. Así, posee métodos para inicializar los gráficos, crear el motor, y ejecutar las acciones necesarias en cada actualización.

Nos hemos basado en los ejemplos de gamelib para la codificación de la clase, de ahí que métodos como `initGfx()` o `updateScreen()` estén en inglés.

- **Declaración**

```
public class MiPanel extends JPanel implements
Runnable
```

- **Importaciones**

```
import java.awt.*;
import java.awt.image.*;
import java.awt.event.*;

import nu.bugbase.gamelib.utils.*;

import javax.swing.JPanel;

import java.util.LinkedList;
```

- **Atributos**

Nombre	Tipo	<i>Descripción:</i>
padre	PruebaAnim	Instancia de la interfaz global.
personajes	LinkedList	Lista de personajes (instancias de Objetos)
CRONO	boolean	Variable para distinguir si estamos seleccionando instancias de la clase Crono (CRONO= true) o Locke (CRONO= false)
bimg	BufferedImage	Doble búfer para la pantalla
bg2	Graphics2D	Contexto gráfico del doble búfer.
background	BufferedImage	Objeto correspondiente a la imagen de fondo
waitingForRepaint	boolean	Indica si estamos esperando a que se produzca el redibujado de la pantalla.
exampleHeight	int	Alto de la pantalla
exampleWidth	int	Ancho de la pantalla

- **Métodos**

<u>Nombre:</u> MiPanel()	<u>Tipo devolución:</u> No aplicable
<u>Argumentos:</u> <ul style="list-style-type: none"> • PruebaAnim _padre: argumento correspondiente a la Interfaz gráfica a la que pertenece el panel. 	
<u>Descripción:</u> Constructor. Fija el atributo “padre” a la referencia pasada como argumento. Además, inicializa la lista de personajes, y fija un oyente para procesar eventos de ratón (en concreto, envía órdenes de movimiento a distintos personajes)	

<u>Nombre:</u> initGfx()	<u>Tipo devolución:</u> void
<u>Argumentos:</u> No tiene	
<u>Descripción:</u> Inicializa los gráficos. Para ello, crea una imagen de doble búfer, el contexto gráfico de la misma, y carga y dibuja la imagen de fondo. Adicionalmente, crea y añade a la lista de personajes los distintos sprites.	

<u>Nombre:</u> updateScreen()	<u>Tipo devolución:</u> void
<u>Argumentos:</u> <ul style="list-style-type: none"> • Graphics g: Contexto gráfico 	
<u>Descripción:</u> Actualiza la pantalla. En primer lugar, comprueba si es necesario efectuar una inicialización de los gráficos. A continuación, pasa a determinar si ha sido el sistema quien ha solicitado la actualización (en cuyo caso, dibuja en la pantalla el contenido del doble búfer), o ha sido el motor gráfico (en este caso, limpia el fondo del doble búfer y actualiza el estado de todos los personajes –los anima si es necesario, y los dibuja después).	

<u>Nombre:</u> paint()	<u>Tipo devolución:</u> void
<u>Argumentos:</u> <ul style="list-style-type: none"> • Graphics g: Contexto gráfico. 	
<u>Descripción:</u> El sistema llama a este método para dibujar la escena en la pantalla. Delegará en el motor gráfico para encargarse de todo – en concreto, el método updateScreen().	

<u>Nombre:</u> update()	<u>Tipo devolución:</u> void
Argumentos: <ul style="list-style-type: none"> Graphics g: contexto gráfico 	
Descripción: El sistema llama a este método para actualizar la escena dibujada en la pantalla. Al igual que paint(), delega en updateScreen.	

<u>Nombre:</u> getPreferredSize()	<u>Tipo devolución:</u> Dimension
Argumentos: No tiene	
Descripción: Devuelve el tamaño correcto de la pantalla, por si algún método lo pregunta. Concretamente, devolverá los valores de (exampleWidth,exampleHeight)	

<u>Nombre:</u> start()	<u>Tipo devolución:</u> void
Argumentos: No tiene	
Descripción: Crea un nuevo hilo y arranca el motor gráfico.	

<u>Nombre:</u> run()	<u>Tipo devolución:</u> void
Argumentos: No tiene	
Descripción: Este método constituye el motor gráfico de la aplicación. Para ello, crea un objeto Ticket, que generará ticks de reloj, a una tasa de 30 por segundo. Determina en primer lugar en qué momento tendrá lugar la próxima actualización de la pantalla, después actualiza el doble búfer. Adicionalmente, llamará al redibujado de la pantalla si no estábamos esperando ya por tal evento. Finalmente, suspende el hilo hasta la próxima actualización.	

<u>Nombre:</u> enviarMuerte()	<u>Tipo devolución:</u> void
Argumentos: No tiene	
Descripción: Recorre la lista de personajes, indicando a aquellos que sean de tipo Aranha que se “mueran” (en realidad, modifica la animación)	

<u>Nombre:</u> moverAranha()	<u>Tipo devolución:</u> void
Argumentos: No tiene	
Descripción: Recorre la lista de personajes, indicando a aquellos que sean de tipo Aranha que se “muevan” (de nuevo, modifica la animación, restaurando la posición de la araña a la inicial)	

<u>Nombre:</u> cambiarSel()	<u>Tipo devolución:</u> void
Argumentos: No tiene	
Descripción: Modifica el flag de selección de personaje	

<u>Nombre:</u> cambiarLockeRects()	<u>Tipo devolución:</u> void
Argumentos: No tiene	
Descripción: Modifica la visualización del área de movimiento para los objetos de tipo Locke, alternando entre no visualizar los rectángulos que marcan el área y visualizarlos. (dicha visualización tiene sentido a efectos de depuración)	

Jerarquía de Sorites

A continuación, detallaremos los métodos y atributos de las clases relacionadas con la jerarquía de sprites de la aplicación de prueba.

Estado.java

Con esta clase, creamos un tipo de datos para encapsular un estado de los posibles del sprite.

- **Declaración:**

```
public class Estado
```

- **Importaciones:**

```
import nu.bugbase.gamelib.sprite.MovePattern;
```


- **Atributos:**

Nombre	Tipo	<i>Descripción:</i>
Tipo	String	Tipo de animación para el estado correspondiente
Nb	String	Nombre del estado
Patron	MovePattern	Patrón de movimiento correspondiente al estado.

- **Métodos:**

<u>Nombre:</u> Estado()	<u>Tipo devolución:</u> No aplicable
<i>Argumentos:</i> <ul style="list-style-type: none"> • String _s: Nombre • String _t: Tipo • MovePattern _m: Patrón de movimiento 	
<i>Descripción:</i> Constructor. Fija los atributos del estado a los valores pasados como argumentos	

<u>Nombre:</u> Estado()	<u>Tipo devolución:</u> No aplicable
<i>Argumentos:</i> <ul style="list-style-type: none"> • String _s: Nombre • String _t: Tipo 	
<i>Descripción:</i> Constructor. Fija los atributos del estado, salvo el patrón de movimiento, que se inicializará a null , a los valores pasados como argumentos.	

<u>Nombre:</u> getTipo()	<u>Tipo devolución:</u> String
<i>Argumentos:</i> No tiene.	
<i>Descripción:</i> Devuelve el tipo de animación del estado.	

<u>Nombre:</u> getNombre()	<u>Tipo devolución:</u> String
<i>Argumentos:</i> No tiene.	
<i>Descripción:</i> Devuelve el nombre del estado.	

<u>Nombre:</u> getPatron()	<u>Tipo devolución:</u> MovePattern
Argumentos: No tiene.	
Descripción: Devuelve el patrón de movimiento asociado al estado.	

<u>Nombre:</u> setNombre()	<u>Tipo devolución:</u> void
Argumentos: <ul style="list-style-type: none"> String _nb: Nombre 	
Descripción: Fija el atributo nb al valor pasado como argumento.	

<u>Nombre:</u> setTipo()	<u>Tipo devolución:</u> void
Argumentos: <ul style="list-style-type: none"> String _t: Tipo 	
Descripción: Fija el tipo de animación al valor pasado como parámetro	

<u>Nombre:</u> setPatron()	<u>Tipo devolución:</u> void
Argumentos: MovePattern _m: Patrón de movimiento	
Descripción: Fija el patrón de movimiento al objeto pasado como argumento.	

LectorXML.java

Esta clase se encarga del adecuado procesamiento de los ficheros XML de configuración de los distintos parámetros necesarios para configurar la animación de los sprites. Se utiliza un parser tipo SAX, por su rapidez de procesamiento.

- **Declaración**

```
public class LectorXML extends DefaultHandler
```

- **Importaciones**

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;
```

```
import java.util.Hashtable;
import java.awt.Rectangle;
```

- **Constantes**

El siguiente conjunto de constantes se utilizará casi exclusivamente para indicar el estado en que se encuentra el parsing del XML.

```
int NADA=0;
int RUTA=1;
int INCX=4;
int INCY=5;
int MOVS=6;
int IDIMG=7;
int IDACC=8;
int COORDX=9;
int COORDY=10;
int ANCHO=11;
int ALTO=12;
```

- **Atributos**

Nombre	Tipo	<i>Descripción:</i>
rutas	String[]	Array que contiene las rutas de las imágenes para un determinado patrón de movimiento.
estados	Hashtable[]	Tabla Hash que almacenará los estados disponibles para un sprite
R	Rectangle[]	Array de rectángulos, para definir un área
rect	Rectangle	Rectángulo en procesamiento
nombreEstado	String	Nombre del estado en procesamiento
tipoEstado	String	Tipo de animación para el estado en procesamiento
IX	float	Incremento en X para el movimiento individual en procesamiento (Asociado a un MovePattern)
IY	float	Incremento en Y para el movimiento individual en proceso. (Asociado a un MovePattern)
movs	int	Nº de frames durante los que se reproducirá un mismo movimiento (Asociado a un MovePattern)
II	int	Índice de la imagen que se visualizará en un movimiento individual concreto (Asociado a un MovePattern)

M	MovePattern	Patrón de movimiento en procesamiento
estado_parsing	int	Estado del parseo del XML (etiqueta actual que se está procesando)
indImg	int	Índice para acceder al array de imágenes
indEst	int	Índice para acceder al array de estados
indAr	int	Índice para acceder al array de rectángulos que delimitarán un área

- **Métodos**

<u>Nombre:</u> startElement()	<u>Tipo devolución:</u> void
Argumentos: <ul style="list-style-type: none"> • String namespaceURI: URI del espacio de nombres, o cadena vacía si no se utiliza procesamiento de espacios de nombres, o el elemento no tiene URI de dicho espacio de nombres. • String localName: Nombre local (sin prefijo) del elemento. • String qualifiedName: Nombre cualificado (con prefijo) del elemento • Attributes atts: Lista de atributos del elemento concreto 	
Descripción: Detecta una etiqueta de inicio para un elemento XML. Dependiendo del elemento, actuaremos de forma distinta en cada caso. Algunos ejemplos: <ul style="list-style-type: none"> • <LIMAGENES tamanho="20">: Crearíamos un nuevo array de String de tamaño 20 (en el atributo "rutas"). Además, fijaríamos indImg a 0. • <IMAGEN> : modificamos la variable estado_parsing a "RUTA". • <AREA tamanho="15">: Instanciamos una array de Rectangle de 15 elementos (atributo "r"), y fijamos indAr a 0. • <ANCHO>: Cambiamos el estado del parseo a "ANCHO". Con esto nos dispondríamos a leer el ancho de un rectángulo 	

<u>Nombre:</u> endElement()	<u>Tipo devolución:</u> void
Argumentos: <ul style="list-style-type: none"> • String namespaceURI: URI del espacio de nombres, o cadena vacía si no se utiliza procesamiento de espacios de nombres, o el elemento no tiene URI de dicho espacio de nombres. • String localName: Nombre local (sin prefijo) del elemento. • String qName: Nombre cualificado (con prefijo) del elemento 	
Descripción: Detecta una etiqueta de fin para un elemento XML. Dependiendo del elemento, actuaremos de forma distinta en cada caso. Algunos ejemplos: <ul style="list-style-type: none"> • </IMAGEN> : Incrementamos indImg. • </ESTADO> : Introducimos un nuevo par (atributo, valor) en la tabla de estados. • </ENTRADA>: Añade un nuevo movimiento individual al patrón de 	

movimiento en curso.

<u>Nombre:</u> characters()	<u>Tipo devolución:</u> void
Argumentos <ul style="list-style-type: none">• char[] buf: Texto contenido entre un par <A>• int offset: Posición de inicio en el buffer.• int len: Longitud del buffer	
Descripción: <p>Leemos los caracteres delimitados entre un par de etiquetas <A>.... En función del estado de parsing actuaremos de modo diferente. Algunos ejemplos:</p> <ul style="list-style-type: none">• RUTA: El texto es una ruta correspondiente a una imagen. La añadimos al array “rutas”.• INCX, INCY: Tratamos de extraer un número en punto flotante de buf, que corresponderá respectivamente a un incremento en X o Y dentro de una de las acciones que constituyen un patrón de movimiento.• COORDX, COORDY: Intentamos leer un número entero de buf. Éste corresponderá a la coordenada X (respectivamente, Y) de la esquina superior izquierda de un rectángulo de los que delimitan un área de movimiento.	

<u>Nombre:</u> error()	<u>Tipo devolución:</u> void
Argumentos: <ul style="list-style-type: none">• SAXParseException e: Excepción de parsing capturada.	
Descripción: <p>Maneja errores no fatales.</p>	

<u>Nombre:</u> warning()	<u>Tipo devolución:</u> void
Argumentos: <ul style="list-style-type: none">• SAXParseException e: Excepción de parsing capturada.	
Descripción: <p>Maneja también errores no fatales, en este caso concreto avisos.</p>	

<u>Nombre:</u> getRutas()	<u>Tipo devolución:</u> String[]
Argumentos: <p>No tiene</p>	
Descripción: <p>Devuelve el array con las rutas de las imágenes</p>	

<u>Nombre:</u> getEstados()	<u>Tipo devolución:</u> Hashtable
<u>Argumentos:</u> No tiene	
<u>Descripción:</u> Devuelve la tabla hash de estados	

<u>Nombre:</u> getArea()	<u>Tipo devolución:</u> Rectangle[]
<u>Argumentos:</u> No tiene	
<u>Descripción:</u> Devuelve el array de rectángulos que conforman un área.	

Objetos.java

Esta clase abstracta sirve como base para nuestra jerarquía de sprites. Además, extiende a un PatternAnimatedSprite, por lo que en principio proporciona ya los métodos principales para dibujarse en la pantalla y animarse adecuadamente. Este comportamiento, sin embargo, podrá redefinirse en las clases hijas si es necesario.

- **Declaración**

```
public abstract class Objetos extends
PatternAnimatedSprite
```

- **Importaciones**

```
import nu.bugbase.gamelib.sprite.*;
import nu.bugbase.gamelib.utils.ImageToolkit;
import java.awt.Rectangle;

import javax.swing.JPanel;

import java.util.Hashtable;

import org.xml.sax.*;
import javax.xml.parsers.*;
import java.io.*;
```

- **Atributos**

Nombre	Tipo	<i>Descripción:</i>
Padre	JPanel	Pantalla en la que se encuentra el sprite. Este atributo se añade por propósitos de comunicación.
L	LectorXML	Objeto que procesa ficheros XML de configuración
Ests	HashTable	Tabla de estados posibles en los que puede encontrarse el sprite
Rects	Rectangle[]	Conjunto de rectángulos que delimitará un área de movimiento. Se utiliza solamente en objetos pertenecientes a la clase Locke, pero realmente deberá ser un atributo de todos los sprites (que luego lo utilicen o no dependerá de la clase)
RUTAXML	String	Ruta en la que se encontrará el archivo XML de configuración

- **Métodos**

<u>Nombre:</u> Objetos()	<u>Tipo devolución:</u> No aplicable
<i>Argumentos:</i> Jpanel _padre: panel que contiene al sprite.	
<i>Descripción:</i> Constructor. Llama a su vez al constructor de PatternAnimatedSprite, con parámetros vacíos, que serán inicializados por las distintas subclases de Objetos. Fija también el panel contenedor del sprite.	

<u>Nombre:</u> inicializa()	<u>Tipo devolución:</u> void
<i>Argumentos:</i> No tiene	
<i>Descripción:</i> <i>Método abstracto. A implementar por las subclases.</i> Inicializa los distintos atributos del sprite.	

<u>Nombre:</u> leerXML()	<u>Tipo devolución:</u> void
<i>Argumentos:</i> No tiene	

Descripción:

Lee el fichero de configuración XML, apoyándose en un objeto lectorXML. Si el análisis del XML tiene éxito, cargará además las imágenes a partir del array “rutas” del objeto lector. Asimismo, cargamos también la tabla de estados y el área de movimiento (constituida por un array de rectángulos leídos del fichero de configuración)

Nombre: moverse()***Tipo devolución:*** void***Argumentos:***

- **int** X: coordenada X del punto al que nos queremos desplazar
- **int** Y: coordenada Y.

Descripción:

Método abstracto. A ser implementado por las subclases.

Fija un destino al que el sprite deberá desplazarse, especificado como un punto bidimensional.

Nombre: getR()***Tipo devolución:*** Rectangle[]***Argumentos:***

No tiene

Descripción:

Devuelve el array de rectángulos que conforman el área de movimiento

Nombre: getLector()***Tipo devolución:*** LectorXML***Argumentos:***

No tiene

Descripción:

Devuelve el objeto lectorXML.

Aranha.java



Ilustración 17: Algunas de las imágenes del sprite, correspondientes a distintos estados

Esta clase fue el primer sprite desarrollado en las pruebas. Las funcionalidades que se han añadido, como desarrollo de algunas de las ideas teóricas expuestas en el apartado 1, son:

- Testeo del uso de patrones de movimiento y estados para gestionar la animación.
- Añadir nuevas formas de manejar colisión contra bordes a las ya existentes en gamelib. En concreto, hemos conseguido que la araña, al moverse a la izquierda y llegar al borde izquierdo de la pantalla, cambie el patrón de movimiento a “moverse a la derecha”, pasando así a desplazarse hacia la derecha. Análogamente ocurre en esta dirección.
- Reaccionar ante eventos generados desde la interfaz de usuario: por ejemplo, al pulsar el botón “Morir” de la interfaz, la araña se sitúa en el centro de la pantalla, cambia el patrón a “muriendo”, y ejecuta una animación diferente a la de movimiento.
- Cambio “automático” de un patrón a otro. En el ejemplo anterior, la araña, una vez finalizada la animación de “muerte”, se vuelve a situar en la esquina inferior derecha y de nuevo se pone a caminar por el borde de la pantalla.

- **Declaración**

```
public class Aranha extends Objetos
```

- **Importaciones**

```
import javax.swing.JPanel;  
import nu.bugbase.gamelib.sprite.*;
```

- **Constantes**

En este caso, las constantes se utilizan para representar los distintos estados del objeto.

```
final static String IZQDA="caminando_oeste";  
final static String DCHA="caminando_este";  
final static String MORIR="muriendo";
```

- **Atributos**

Esta clase posee únicamente los atributos heredados de su superclase.

- **Métodos**

<u>Nombre:</u> Aranha()	<u>Tipo devolución:</u> No aplicable
<u>Argumentos:</u> <ul style="list-style-type: none"> • Jpanel _padre: Panel que contiene al sprite 	
<u>Descripción:</u> Constructor. Llama al constructor de la superclase con el parámetro pasado como argumento	

<u>Nombre:</u> inicializa()	<u>Tipo devolución:</u> void
<u>Argumentos:</u> No tiene	
<u>Descripción:</u> Realiza funciones de inicialización. Así, llama a analizar el fichero XML, e inicializa el patrón de movimiento (En este caso, el patrón corresponderá al estado “moverse a la izquierda”)	

<u>Nombre:</u> handleBorderAction()	<u>Tipo devolución:</u> void
<u>Argumentos:</u> No tiene	
<u>Descripción:</u> Determina la acción a tomar con respecto a los bordes de movimiento, si excede alguno de ellos, en función de la estrategia de resolución que hayamos tomado. En los casos por defecto, llama al método de la superclase (que, subiendo en la jerarquía, se encuentra implementado en AnimatedSprite). En otro caso, modifica el patrón de movimiento, de forma que se invierta la dirección del mismo. Es decir, si el estado actual es “moverse a la izquierda”, y nos salimos por el borde izquierdo de la pantalla, el método cargaría el patrón correspondiente al estado “moverse a la derecha”, y viceversa.	

<u>Nombre:</u> matar()	<u>Tipo devolución:</u> void
<u>Argumentos:</u> No tiene	
<u>Descripción:</u> Cambia el patrón de movimiento al correspondiente al estado “muriendo”, y reubica el sprite.	

<u>Nombre:</u> mover()	<u>Tipo devolución:</u> void
Argumentos: No tiene	
Descripción: Restaura el patrón de movimiento al correspondiente al estado “moverse a la izquierda”	

<u>Nombre:</u> animate()	<u>Tipo devolución:</u> void
Argumentos: No tiene	
Descripción: Modificaría los parámetros del sprite (posición, imagen dibujada,...)de acuerdo al tipo de animación. En este caso, se limita a llamar al método de la superclase, añadiendo una comprobación sobre si, en el estado “muriendo”, se ha llegado al final de la animación, con el fin de restaurar el patrón de movimiento a “moverse a la izquierda”	

<u>Nombre:</u> moverse()	<u>Tipo devolución:</u> void
Argumentos: <ul style="list-style-type: none"> • int X: coordenada X del destino al que desplazarse. • int Y: coordenada Y del destino al que desplazarse. 	
Descripción: No hace nada.	

Knuckles.java



Ilustración 18: Algunas de las imágenes del sprite, correspondientes a algunos de sus estados.

A la araña le siguió esta clase. El objetivo era implementar un sprite que se desplazara con una dirección concreta, indicada a partir de una pulsación del ratón en la pantalla y según una recta trazada de la posición del sprite a las coordenadas dadas por el usuario con el clic del ratón. Además, se deseaba incluir distintos métodos de animación del sprite. Todo esto, en un principio, se pensó de forma que pudieramos variar entre un `PatternAnimatedSprite` y un `VectorAnimatedSprite`. No obstante, esto conllevaría modificar la jerarquía para nuestros sprites, y complicarla innecesariamente. Finalmente, se optó por utilizar en cualquier caso un patrón de movimiento, simulando la funcionalidad de un sprite animado con un vector por medio de dos estados, cuyos patrones permitirían modificar los incrementos en X e Y de cada acción individual. Así, tendremos dos tipos de estados:

- Patrón, que incluye un patrón de movimiento clásico.

- Dirección, que modifica los incrementos de acuerdo con la dirección dada.

Para llevar esto a cabo, fue necesario añadir a la clase MovePattern el siguiente método:

```
void setDxDy(float _incX, float _incY, int index)
```

Lo que hace es modificar la acción especificada por index en el patrón, sustituyendo dx[index] y dy[index] por los valores pasados como argumentos, _incX e _incY, respectivamente.

- **Declaración.**

```
public class Knuckles extends Objetos
```

- **Importaciones**

```
import nu.bugbase.gamelib.sprite.*;
import nu.bugbase.gamelib.utils.*;
import javax.swing.JPanel;
```

- **Atributos**

Tampoco esta clase dispone de atributos propios, únicamente aquellos heredados de la superclase, por lo que no nos detendremos a listarlos.

- **Métodos**

<u>Nombre:</u> Knuckles()	<u>Tipo devolución:</u> No aplicable
<u>Argumentos:</u> <ul style="list-style-type: none"> • Jpanel _padre: panel que contiene al sprite. 	
<u>Descripción:</u> Constructor. Llama al constructor de la superclase, con el parámetro pasado como argumento.	

<u>Nombre:</u> inicializa()	<u>Tipo devolución:</u> void
<u>Argumentos:</u> No tiene	
<u>Descripción:</u> Inicialización. Lee el fichero XML de configuración propio del sprite, carga un patrón por defecto y realiza alguna función adicional.	

<u>Nombre:</u> moverse()	<u>Tipo devolución:</u> void
<u>Argumentos:</u> <ul style="list-style-type: none"> • int X: coordenada X del objetivo. • int Y: coordenada Y del objetivo. 	
<u>Descripción:</u> Fija la dirección según la cual desplazarse, determinado por medio de un punto (X,Y). Comprueba en primer lugar el sentido del movimiento de acuerdo al eje X, para cargar correctamente uno u otro patrón (“derecha” “izquierda”). A continuación traza una recta desde la posición del sprite a la del objetivo, cuya dirección en grados se utilizará para modificar la dirección de cada acción del patrón.	

Crono.java



Ilustración 19: Algunas imágenes del sprite.

Esta clase se desarrolló para cambiar la forma de dirigirse al destino del sprite que se había implementado en la clase anterior. En lugar de seguir una determinada dirección, la idea es descomponer el movimiento en dos:

- En primer lugar, un desplazamiento horizontal, hasta llegar a la X del destino, y
- en segundo lugar, un desplazamiento vertical desde esa posición, hasta llegar al destino efectivo.

Adicionalmente, y a diferencia de la clase anterior, en este caso una vez alcanzado el destino determinado con el ratón, el sprite modificaría su estado, y con ello su animación correspondiente, a una situación de reposo.

- **Declaración**

```
public class Crono extends Objetos
```

- **Importaciones**

```
import nu.bugbase.gamelib.sprite.*;
import javax.swing.JPanel;
```

- **Constantes**

En este caso, se emplean las siguientes constantes, para determinar el nombre de los distintos estados del sprite.

```
final static String REPOSO="parado";
final static String ESTE="caminando_este";
final static String OESTE="caminando_oeste";
final static String NORTE="caminando_norte";
final static String SUR="caminando_sur";
```

- **Atributos**

En este caso, además de los atributos heredados, disponemos de los siguientes:

Nombre	Tipo	<i>Descripción:</i>
TopeX	int	Coordenada X del destino
TopeY	int	Coordenada Y del destino
Actual	String	Nombre del estado actual.

- **Métodos**

<u>Nombre:</u> Crono()	<u>Tipo devolución:</u> No aplicable
<i>Argumentos:</i> <ul style="list-style-type: none"> • JPanel _padre: Panel que contiene al sprite 	
<i>Descripción:</i> Constructor. Llama al constructor de la superclase, con el parámetro pasado como argumento.	

<u>Nombre:</u> inicializa()	<u>Tipo devolución:</u> void
<i>Argumentos:</i> No tiene	
<i>Descripción:</i> Diversas acciones de inicialización, principalmente la lectura del fichero de configuración.	

<u>Nombre:</u> moverse()	<u>Tipo devolución:</u> void
<i>Argumentos:</i> <ul style="list-style-type: none"> • int X: coordenada X del destino • int Y: coordenada Y del destino 	
<i>Descripción:</i> Inicializa un movimiento desde la posición actual a la dada por (X,Y). En primer lugar, determina los valores de topeX, topeY a partir de (X,Y), para ubicar correctamente al sprite una vez que llegue al destino. A continuación, modifica el estado desde el que partirá el movimiento. Así, si la coordenada X no coincide ya con la del destino, nos desplazaremos horizontalmente en el sentido adecuado (cargando el patrón correspondiente ese estado); de coincidir, pasaríamos directamente a un movimiento vertical.	

Nombre: animate()	<u>Tipo devolución:</u> void
Argumentos: No tiene	
Descripción: Animamos el sprite, determinando su próximo movimiento. Así, si no nos encontramos en un estado de reposo, dependiendo del destino especificado y el estado actual, cargaremos el patrón de movimiento adecuado. Por ejemplo, si estábamos desplazándonos hacia la izquierda, y comprobamos que ya hemos alcanzado la coordenada X del destino, pasaríamos a determinar el sentido correcto del movimiento en vertical (o directamente, al estado de reposo si ya nos encontramos en el destino) Por último, llamaría al método animate() de la superclase, para llevar a cabo las demás tareas precisas para animar el sprite.	

Locke.java



Ilustración 20: Algunos sprites del personaje

Esta clase es una ampliación de la clase Crono (sin embargo, no hereda de ella), en la que el objetivo principal desarrollado fue modificar el área de movimiento, de forma que no nos restringiéramos a una zona rectangular. Para ello, se delimita el área permitida por medio de un conjunto de rectángulos.

Importante: El área de rectángulos debe especificarse teniendo en cuenta las siguientes directrices:

- ❖ El área debe extenderse de forma que ocupe toda la pantalla.
- ❖ El primer y último rectángulos comprenden el área de pantalla sobrante con respecto a las coordenadas extremas que queremos tenga la región permitida.
- ❖ Los demás rectángulos deben darse por parejas, con el mismo ancho, de forma que el primero de ellos ocupe la parte norte del tramo concreto del área, y el segundo la parte sur.

Para ayudar a comprender estas directrices, véase el siguiente gráfico:

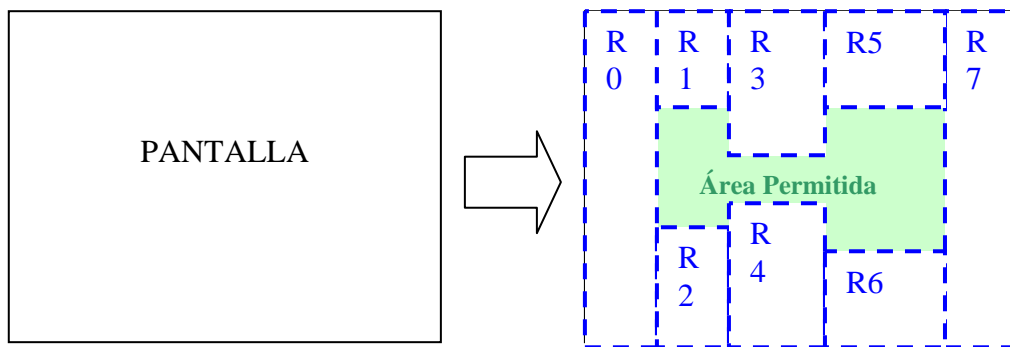


Ilustración 21: Definición del área permitida de movimiento

```
public class Locke extends Objetos
```

- **Importaciones**

```
import java.awt.Rectangle;
import java.awt.Graphics2D;
import javax.swing.JPanel;

import nu.bugbase.gamelib.sprite.AnimatedSprite;
import nu.bugbase.gamelib.sprite.MovePattern;
```

- **Constantes**

Determinan el posible estado actual del sprite.

```
final static String REPOSO="parado";
final static String ESTE="caminando_este";
final static String OESTE="caminando_oeste";
final static String NORTE="caminando_norte";
final static String SUR="caminando_sur";
```

- **Atributos**

Además de los heredados, tenemos los siguientes atributos:

Nombre	Tipo	<i>Descripción:</i>
TopeX	int	Coordenada X del destino
TopeY	int	Coordenada Y del destino
indice	int	Indice del array de rectángulos prohibidos que señala el rectángulo concreto contra el que se ha colisionado.
Actual	String	Nombre del estado actual en que se encuentra el sprite
verRects	boolean	Determina si se visualizarán los rectángulos prohibidos o no en la

		pantalla.
prohibidos	Rectangle[]	Área prohibida para el sprite.

- **Métodos**

<u>Nombre:</u> Locke()	<u>Tipo devolución:</u> No aplicable
Argumentos: <ul style="list-style-type: none"> • JPanel _padre: panel que contiene al sprite. 	
Descripción: Constructor. Llama al constructor de la superclase, pasándole como parámetro el argumento recibido.	

<u>Nombre:</u> inicializar()	<u>Tipo devolución:</u> void
Argumentos: No tiene	
Descripción: Acciones de inicialización, entre las que destaca la lectura del XML de configuración. Fija también el patrón de movimiento de acuerdo con el estado actual, y el conjunto de rectángulos prohibidos.	
<u>Nombre:</u> setArea()	<u>Tipo devolución:</u> void
Argumentos: <ul style="list-style-type: none"> • Rectangle[] _borde: Área de rectángulos prohibidos a fijar. 	
Descripción: Invalida el borde rectangular existente por defecto, y fija el área de movimiento, delimitada por un conjunto de rectángulos prohibidos.	

<u>Nombre:</u> setBorder()	<u>Tipo devolución:</u> void
Argumentos: <ul style="list-style-type: none"> • Rectangle rect: Región rectangular a fijar. 	
Descripción: Invalida el área dada por medio de un conjunto de rectángulos, y fija un borde rectangular	

<u>Nombre:</u> dibujaRectangulos()	<u>Tipo devolución:</u> void
Argumentos: <ul style="list-style-type: none"> • Graphics2D g: Contexto gráfico. 	
Descripción: Dibuja los rectángulos que determinan la región de movimiento. Tiene propósitos de depuración, principalmente.	

<u>Nombre:</u> isOutsideBorder()	<u>Tipo devolución:</u> boolean
Argumentos: No tiene	
Descripción: Determina si el sprite, en su posición actual, excede o bien el borde rectangular impuesto (ante lo que llamaría al isOutsideBorder() de la superclase), o interseca con alguno de los rectángulos prohibidos. En este caso, ajustaría el atributo “índice” dependiendo del estado actual (que puede estar relacionado con el sentido de movimiento) y del rectángulo contra el que se ha detectado intersección. Esto tiene su razón de ser en que, dependiendo del sentido en el que nos estemos moviendo, es posible que la colisión real tenga lugar con el rectángulo “siguiente en el área” al que estamos procesando, por lo que el índice del primer rectángulo con el que existe intersección no es correcto.	

<u>Nombre:</u> animate()	<u>Tipo devolución:</u> void
Argumentos: No tiene	
Descripción: Animamos el sprite, determinando su próximo movimiento. Así, si no nos encontramos en un estado de reposo, dependiendo del destino especificado y el estado actual, cargaremos el patrón de movimiento adecuado. Por ejemplo, si estábamos desplazándonos hacia la izquierda, y comprobamos que ya hemos alcanzado la coordenada X del destino, pasaríamos a determinar el sentido correcto del movimiento en vertical (o directamente, al estado de reposo si ya nos encontramos en el destino) Por último, llamaría al método animate() de la superclase, para llevar a cabo las demás tareas precisas para animar el sprite.	

<u>Nombre:</u> moverse()	<u>Tipo devolución:</u> void
Argumentos: <ul style="list-style-type: none"> • int X: coordenada X del destino. • int Y: coordenada Y del destino. 	
Descripción: Inicializa el movimiento a realizar por el sprite, para desplazarse hasta el punto dado por (X,Y), del mismo modo que tenía lugar en la clase Crono.	

<u>Nombre:</u> handleBorderAction()	<u>Tipo devolución:</u> void
Argumentos: No tiene	

Descripción:

Gestiona una posible colisión contra los bordes del área de movimiento. Si el área es rectangular, delega en el método del mismo nombre de las superclases.

Nombre: stopOnBorder()

Tipo devolución: void

Argumentos:

No tiene

Descripción:

Determina las acciones a tomar para detenerse al llegar al borde de la pantalla. En concreto, calculamos la posición del sprite, y fijamos el desplazamiento a realizar en cada animación a 0 en ambos ejes (para que se detenga). Por último, fija el estado actual al de “reposo”.

Nombre: draw()

Tipo devolución: void

Argumentos:

- Graphics2D g

Descripción:

Llama a dibujaRectangulos() o no, dependiendo del valor de la variable verRects, y a continuación delega en el método draw () de la superclase para dibujar el sprite.

Nombre: cambiaVerRects()

Tipo devolución: void

Argumentos:

No tiene

Descripción:

Invierte el valor de la variable verRects.

4.2.3. Conclusiones

Finalizado el desarrollo de las distintas pruebas con sprites, se concluyó que una gran parte de las ideas, e incluso parte del código, resultarían en principio adaptables y reutilizables a la hora de integrar animación en la AGM. En principio, la estructura de los sprites se adaptaría a la de la clase Objetos, añadiendo la funcionalidad de los distintos sprites más concretos desarrollados. Podría especializarse el funcionamiento de los sprites, atendiendo a la naturaleza de los objetos en pantalla (por ejemplo, si no deseásemos que un Objeto No Personaje (ONP) se moviera, pero sí estar animado).

También podría reutilizarse código de la clase correspondiente a la “pantalla” de la aplicación, sobre la cual se dibujan los sprites.

Se verá esto con más detenimiento en el apartado 5.

4.3. **Gamelib**

Gamelib es un paquete gratuito con clases útiles para el desarrollo de juegos Java. Se trata de un proyecto de código abierto.

La librería se estructura sobre cuatro paquetes:

- *nu.bugbase.gamelib.background*: Únicamente posee una clase, que genera como fondo un campo de estrellas.
- *nu.bugbase.gamelib.sprite*: Comprende aquellas clases que implementan las distintas variaciones básicas sobre un sprite vacío, este último también incluido.
- *nu.bugbase.gamelib.text*: Incluye clases relacionadas con el manejo de textos, para su utilización como subtítulos, o “texto deslizante”.
- *nu.bugbase.gamelib.utils*: Contiene un conjunto de utilidades para la interacción con el usuario vía joystick, manejo de sonido, o generación de ticks de reloj.

Durante la fase de investigación, y presumiblemente en la fase de integración (presente o futura) de la animación con la aplicación principal, únicamente han resultado de utilidad el paquete “sprite”, y en menor medida “utils”. Será sobre el primero de estos en los que se hará hincapié a continuación, puesto que constituyen la base sobre la que se han implementado las pruebas, y sobre la que se integrará con el proyecto final.

Por último, se expondrán brevemente las conclusiones a las que se llegó con el estudio previo de la librería.

4.3.1. *nu.bugbase.gamelib.sprite*

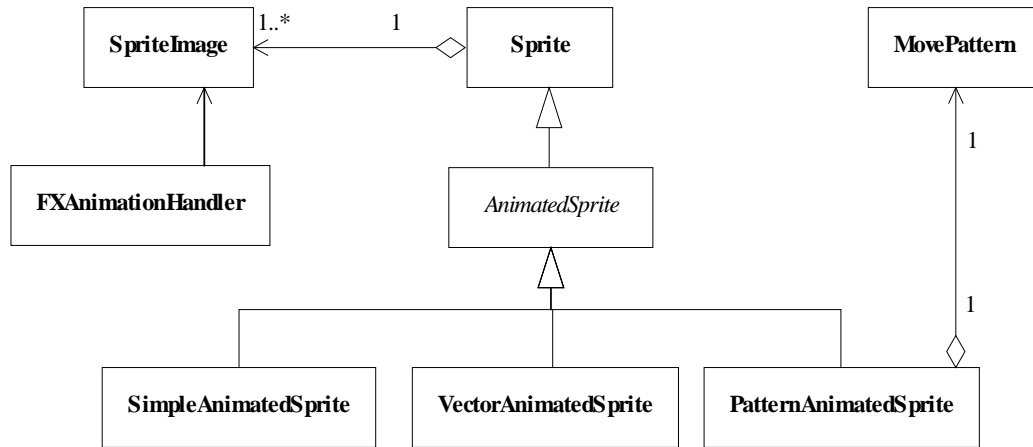


Ilustración 22: Diagrama de clases simplificado para el paquete

Este paquete reúne las siguientes clases:

Sprite

Esta clase, que constituye la cima de la jerarquía, representa un sprite en la pantalla. Se encarga además de manejar el dibujo del propio sprite, así como los tests de colisión.

Es posible que el sprite contenga una única imagen (encapsulada en una instancia de *SpriteImage*), o un conjunto de ellas, y poder dibujar en cada momento la que convenga (Esto será especialmente útil cuando se empleen sprites animados, pues permitirá conferir un mayor realismo a los movimientos).

Proporciona métodos para fijar la posición de la imagen, modificar el estado activo/inactivo (y actuar en consecuencia a la hora de dibujar la imagen en pantalla), desplazarse, o bien fijar los límites que delimitan el área en la que se puede situar el sprite (una vez más, esto tiene una gran utilidad con sprites animados, para evitar por ejemplo que se excedan de los límites de la pantalla y “desaparezcan” sin más). En relación con esta última capacidad, también es posible conocer el borde contra el cual puede estar a punto de excederse.

SpriteImage

Encapsula un objeto de tipo *Image* y datos específicos del sprite, como “hotspots”, la máscara, y valores de traslación.

Para la mayor parte de las operaciones que se realizan sobre esta clase, como pueda ser recortar la imagen, o generar la máscara (útil para la detección de colisiones, si se quiere una precisión elevada en dicha detección), se requiere además de la clase *ImageToolkit*, del paquete de utilidades.

AnimatedSprite

Esta clase abstracta se utiliza como base para todas aquellas clases que representen sprites animados.

Además de la funcionalidad de la clase *Sprite*, proporciona diversos métodos como respuesta ante una colisión con los bordes que delimitan el rectángulo de la pantalla por el que puede moverse el sprite, como pueda ser rebotar contra ellos, detenerse, o desaparecer por un borde para reaparecer por el opuesto.

Por último, incluye un método abstracto, *animate()* que deberán implementar las clases que hereden de ella para determinar el movimiento a realizarse y simular así la animación.

SimpleAnimatedSprite

Representa el sprite animado más simple posible. Esto es, fijados incrementos en el eje X y el eje Y, calcula el siguiente movimiento en función de la posición actual de la imagen actual como

$$\begin{aligned} \text{nuevaX} &= \text{viejaX} + \text{incrementoX} \\ \text{nuevaY} &= \text{viejaY} + \text{incrementoY} \end{aligned}$$

Pueden modificarse manualmente los incrementos en X e Y.

VectorAnimatedSprite

En este caso, la clase se sirve de un vector para calcular la dirección sobre la que se desplazará el sprite.

La transformación que se llevará a cabo al animar la imagen será:

$$\begin{aligned} \text{nuevaX} &= \text{viejaX} + \text{velocidad} * \text{incrementoX} \\ \text{nuevaY} &= \text{viejaY} + \text{velocidad} * \text{incrementoY} \end{aligned}$$

El vector que guía el movimiento puede especificarse de distintas formas:

- A partir de un punto(x,y).
- A partir de un ángulo, dado en grados.
- A partir de otro sprite. En este caso la dirección vendría dada por el vector normalizado que tiene por origen la posición de nuestro sprite, y por extremo la del objetivo.

PatternAnimatedSprite, y MovePattern

En esta clase, el sprite se mueve con ayuda de un patrón de movimiento.

Un patrón es una secuencia de movimientos individuales. Para cada uno de estos es preciso detallar:

- dx – Distancia sobre el eje X que se desplazará en este movimiento el sprite.
- dy – Análogamente para el eje Y.
- moves – Durante cuántos marcos de animación utilizaremos estos valores, antes de pasar a la siguiente entrada en la secuencia.

Adicionalmente existen otros dos parámetros posibles:

- **id** – Cuando el sprite consta de varias imágenes, especifica cuál de todas ellas se utilizará en el movimiento actual.
- **action** – Fija una acción especificada por el usuario.

También es posible determinar el índice en la secuencia de movimientos desde el cual se repetirá el patrón.

Como se puede comprobar, esta clase proporciona una gran versatilidad a la hora de especificar movimientos, así como control sobre los mismos y la imagen/imágenes que constituirán la animación. Por todo esto constituirá la base sobre la cual se definirán nuestros propios sprites.

FXAnimationHandler

Esta clase puede manejar un conjunto múltiple de efectos de animación, tales como explosiones. No diremos mucho más sobre esta clase.

4.3.2. *nu.bugbase.gamelib.utils*

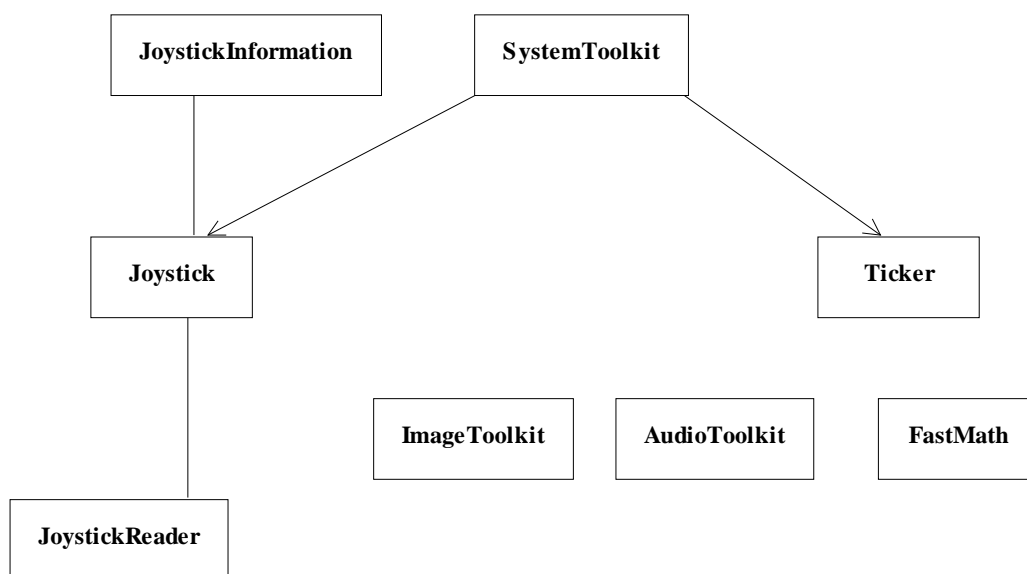


Ilustración 23: Diagrama de clases simplificado para el paquete

Este paquete contiene clases de utilidad. Para el caso que nos ocupa, la mayor parte no resultan necesarias, por lo que las comentaremos muy brevemente, salvo aquellas que revistan un mayor interés.

AudioToolkit

Contiene un conjunto de métodos estáticos, que permiten cargar y reproducir clips de audio.

FastMath

Contiene algoritmos geométricos, utilizando tablas precalculadas de senos y cosenos para lograr una mayor velocidad. Así, podemos calcular rápidamente el seno o el coseno de un ángulo dado en grados, o conocer el ángulo que forma un vector.

ImageToolkit

Incluye una serie de métodos estáticos para manejar instancias de la clase Image de la awt.

Así, podemos cargar imágenes a partir de un array de pixels (modo ARGB), de un recurso, generar la máscara de una imagen (un mapa de bits resaltando los pixels no transparentes de la misma), aplicar diversas transformaciones, como rotaciones y traslaciones, o recortarlas

Joystick

Define un objeto joystick.

JoystickInformation

Contiene información de calibración acerca de una instancia de Joystick.

JoystickReader

Un objeto perteneciente a esta clase lee y convierte los valores de salida de Joystick.

SystemToolkit

Esta clase ofrece un conjunto de métodos estáticos que posibilitan crear objetos del sistema. En realidad podría considerarse una factoría de instancias de Joystick, o de Ticker.

Ticker

Define un objeto Ticker, que se utilizará para generar ticks de reloj, o enviar al proceso a dormir durante un cierto período de tiempo.

Objetos de esta clase se emplearán para actualizar la pantalla (o el panel que representará a la pantalla), haciendo que cada sprite se anime (cuando proceda) y dibuje adecuadamente.

4.3.3. Conclusiones

La observación de los ejemplos incluidos con la librería, el código de los mismos y la inspección del propio código fuente de las clases de la librería, suministrado junto al fichero *.jar y los ejemplos, resultó suficientemente convincente como para decidir emplearla como base para las pruebas y la posterior integración. La librería está bien

documentada, resulta sencilla de utilizar y modificar, y sus capacidades pueden ser extendidas a voluntad del programador.

La descripción de las clases ofrecida en los epígrafes anteriores es un simple resumen de su funcionalidad, y puede resultar insuficiente. Sin embargo, escapa a los objetivos del presente documento realizar una exposición detallada acerca de todos los pormenores de la librería. Para obtener una información más completa acerca de la librería, se recomienda consultar la especificación de la API de la misma.

Tanto la API, como información adicional acerca del proyecto gamelib, descargas, etc., pueden consultarse en la página web del grupo de desarrollo:

http://home.swipnet.se/~w-56899/gamelib_main.html (en inglés)

4.4. Ficheros de Configuración

Para determinar la lista de imágenes a cargar, de estados, con sus correspondientes patrones de movimiento, o del área prohibida que delimita la región de movimiento, se debe leer un fichero XML específico para cada tipo de sprite concreto. La estructura del mismo se detalla a continuación. Adicionalmente, se proporcionará en un segundo subapartado una serie de comentarios destinados a ampliar los parámetros que se leen del XML.

4.4.1. Estructura de un XML de configuración

En primer lugar, veremos la estructura de los documentos examinando la dtd asociada.

Básicamente, un xml para configurar las diferentes características del sprite constará de:

```
<!ELEMENT ANIMACION (LIMAGENES,LESTADOS,AREA?)>
```

- **Limágenes** contiene una lista con las rutas de las imágenes a cargar.
- **Lestados**, por su parte, posee una lista de estados. Cada uno de estos almacena un patrón de movimiento, y además el nombre del estado y el tipo de animación que se realizará (este último atributo, en principio, está en desuso)
- **Area** es un elemento opcional, que codifica una lista de rectángulos. Estos se utilizarán para delimitar la región por la que puede moverse un sprite.

Estas tres listas poseen un atributo “tamanho”, para determinar el número de elementos de cada una de ellas.

Prosiguiendo con la dtd, veremos ahora cómo se codifica el estado de un objeto, y en especial el patrón a él asociado.

```
<!-- estado -->
<!ELEMENT ESTADO (PATRON)>
<!--ATTLIST ESTADO
      nombre CDATA #REQUIRED
      tipo CDATA #REQUIRED -->
<!-- Patron de movimiento -->
<!ELEMENT PATRON (ENTRADA)+>
<!--definicion de las entradas-->
<!ELEMENT ENTRADA (INCX,INCY,MOVS,IDIMG,IDACC)>
```

```

<!ELEMENT INCX (#PCDATA)>
<!ELEMENT INCY (#PCDATA)>
<!ELEMENT MOVS (#PCDATA)>
<!ELEMENT IDIMG (#PCDATA)>
<!ELEMENT IDACC (#PCDATA)>

```

Vemos que un patrón consta de una serie de entradas. Cada una de ellas constará de los campos de un MovePattern, que ya se explicaron anteriormente.

Por último, veremos cómo se determinan los rectángulos que delimitan el área de movimiento:

```

<!--Rectangulo -->
<!ELEMENT RECTANGULO (X,Y,ANCHO,ALTO)>
<!ELEMENT X (#PCDATA)>
<!ELEMENT Y (#PCDATA)>
<!ELEMENT ANCHO (#PCDATA)>
<!ELEMENT ALTO (#PCDATA)>

```

Como ejemplo real de documento XML, veremos fragmentos del xml que leen los sprites de la clase Locke, que, a diferencia de los demás, posee área no rectangular de movimiento.

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE ANIMACION SYSTEM "xml/anim5.dtd" >
<ANIMACION>

<!-- lista de imágenes -->

<LIMAGENES tamanho="15">
....
<IMAGEN>imagenes/rlocke003.gif</IMAGEN>
<IMAGEN>imagenes/nlocke001.gif</IMAGEN>
.....
<IMAGEN>imagenes/elocke003.gif</IMAGEN>
<IMAGEN>imagenes/wlocke001.gif</IMAGEN>
... .
</LIMAGENES>

```

Así se especifican las rutas con las imágenes

```

<LESTADOS tamanho="5">
<ESTADO nombre="parado" tipo="patron">
<PATRON>
    <ENTRADA>
        <INCX>0</INCX>
        <INCY>0</INCY>

```

```

        <MOVS>3</MOVS>
        <IDIMG>0</IDIMG>
        <IDACC>0</IDACC>
    </ENTRADA>
    <ENTRADA>
        <INCX>0</INCX>
        <INCY>0</INCY>
        <MOVS>3</MOVS>
        <IDIMG>1</IDIMG>
        <IDACC>0</IDACC>
    </ENTRADA>
    <ENTRADA>
        <INCX>0</INCX>
        <INCY>0</INCY>
        <MOVS>3</MOVS>
        <IDIMG>2</IDIMG>
        <IDACC>0</IDACC>
    </ENTRADA>
</PATRON>
</ESTADO>
...Otros estados....
</LESTADOS>

```

Así, los estados con sus correspondientes patrones de movimiento

```

<AREA tamanho="8">
    <RECTANGULO>
        <X>0</X>
        <Y>0</Y>
        <ANCHO>160</ANCHO>
        <ALTO>600</ALTO>
    </RECTANGULO>

    <RECTANGULO>
        <X>160</X>
        <Y>0</Y>
        <ANCHO>160</ANCHO>
        <ALTO>150</ALTO>
    </RECTANGULO>

    <RECTANGULO>
        <X>160</X>
        <Y>450</Y>
        <ANCHO>160</ANCHO>
        <ALTO>150</ALTO>
    </RECTANGULO>
...Resto de los rectángulos....
</AREA>
</ANIMACION>

```

Y, por último, el conjunto de rectángulos que delimitan un área

4.4.2. *Ampliaciones a realizar sobre los XML*

A la vista del código del programa de pruebas, vemos que algunos parámetros de los sprites, como la respuesta ante colisión contra un borde, el propio borde en el caso por defecto, o la posición del sprite, se determinan directamente en dicho código, lo que está bien en un programa de pruebas, y con unos pocos sprites en pantalla, pero que en una aplicación real, además de poco mantenible, puede resultar engorroso. Por esto, sería muy recomendable añadir los valores de tales parámetros en el XML. Así, en el futuro se deberían añadir, entre otros:

- Posición inicial del sprite: Coordenadas X,Y
- Patrón a cargar por defecto en la inicialización. Valdría con el nombre del estado.
- El borde de movimiento, para los casos en que para éste valga con un área rectangular (NOTA: Esto, aunque no se ha probado, podría simularse con el área ya existente, y cuatro rectángulos.)
- Tipo de respuesta ante colisión contra los bordes.
- Tipo de respuesta ante colisión contra otros sprites.
- Pausa de animación, es decir, el número de frames a esperar hasta que continúe la animación.

4.5. *Integración con AGM*

Por falta de tiempo, y problemas surgidos en la detección de colisión contra los bordes, cuya solución llevó más tiempo del deseable, la integración de la animación en la aplicación final no ha podido realizarse más que parcialmente (se ha incluido un nuevo paquete con las clases ya modificadas). En el momento actual en que se redactó este documento, se estaba realizando un proceso de depuración de las clases integradas, para así dotar a la aplicación de una animación ya integrada en sus aspectos más básicos.

Sin embargo, se proporcionará en el presente apartado una serie de recomendaciones a seguir en el momento de continuar dicha integración:

Pantalla de la aplicación:

Las funciones de la pantalla principal (la ventana del juego) en la AGM las desempeña la clase Bot. En este caso, bastaría en principio con modificar la clase adecuadamente, añadiendo la mayoría de los métodos de la ventana principal del programa de pruebas.

Pendiente de implementación:

ActualizarPantalla() requiere de algunas modificaciones (vienen indicadas como texto comentado).

ObjetoEnPantalla:

Esta clase será la candidata a extender las capacidades de un sprite. En principio, a diferencia de lo que ocurría en el programa de pruebas, esta clase no será abstracta, y todos los objetos que se representen serán instancias de ella, es decir, tendrán todas la misma estructura. Si resultara más recomendable que, por ejemplo, objetos y personajes dispusieran de una estructura más heterogénea, podría definirse una jerarquía de clases que extendieran a ésta.

Pendiente de implementación:

Restan por añadir métodos de cambio de estado, correcciones en el método moverse(X,Y) y tal vez algunas ampliaciones.

InterfazGrafica:

Adaptar esta clase a la nueva clase ObjetoEnPantalla no parece tampoco en principio una tarea complicada. Principalmente, consistiría en revisar las apariciones de instancias de ObjetoEnPantalla y adecuarlas a los posibles cambios.

ClienteJugador:

Modificar esta clase consistiría también en una revisión de lo ya existente. Debe tenerse cuidado con los cambios de estado y la respuesta ante distintas acciones.

Pendiente de Implementación:

Además, si queremos, por ejemplo, que al ejecutar una acción “Coger un objeto X” se visualice al personaje desplazándose hasta la posición del objeto y que no se ejecute la acción en sí de coger el objeto hasta que el personaje ha llegado, deberíamos incluir mecanismos de comunicación entre los objetos en pantalla existentes y la aplicación, de forma que cuando el sprite se encuentre suficientemente cercano al objeto envíe una señal a la aplicación (que tendría que ponerse de algún modo “a la escucha”, y permitir dedicarse mientras tanto a otra cosa, como el intercambio de mensajes con otros jugadores, o directamente “dormirse”) para que se ejecute la acción de forma efectiva. Aquí es donde pueden presentarse problemas.

Serán precisos también mecanismos de comunicación entre instancias de la aplicación, para permitir la correcta actualización de la interfaz de usuario de todos los usuarios. Es decir, que un PJ efectúe un movimiento de X a Y, y esta acción se vea reflejada en las pantallas de todos los demás jugadores

Asimismo, debería comprobarse la utilidad de la máscara.

Resto de clases

Por el momento, no se ve que existan problemas con las demás clases del proyecto. En caso contrario, habría que estudiarlos con un mayor detenimiento.

4.6. Cómo crear una animación para un personaje en AGM

Una vez esté integrada la animación en la aplicación final, el siguiente paso lógico sería definir las animaciones para los personajes (y en general para cualquier objeto).

Resumidos, se muestran a continuación los pasos a seguir:

1. Establecer los posibles estados para los que necesitemos una animación.
2. Determinar las imágenes de que dispondremos para dotar de “vida” al personaje.
3. Asociar a cada estado posible las imágenes correspondientes al mismo, indicando además cuánto debería moverse según los ejes X e Y, y durante cuántos marcos de animación.
4. Si el objeto se mueve, puede ser conveniente establecer el área permitida, especialmente si dicho movimiento no está preestablecido (es decir, si el objeto

- se moverá aleatoriamente, o en respuesta a un clic de ratón, por ejemplo). En este caso, habrá que configurar la región.
5. Con todos estos datos, crear el fichero de configuración XML, y guardarlo en su ubicación correspondiente.
 6. Por último, si se quieren añadir nuevas funcionalidades a ese objeto que no se traduzcan directamente en un cambio de estado, es posible que hayan de ser añadidos nuevos métodos en la clase ObjetoEnPantalla, así como en las asociadas a la interfaz gráfica (si procede). Por ejemplo, si quisiéramos que al pinchar y arrastrar con el ratón sobre la pantalla principal, un objeto “lápiz” trazara una estela, necesitaríamos incluir un método “estelaLapiz()” en ObjetoEnPantalla, e implementar un método OnMouseDragged() (o similares) que llamase a estelaLapiz() en el objeto concreto, con los parámetros necesarios.

Veremos en el siguiente epígrafe un ejemplo práctico, donde se aplicarán estos pasos para crear la animación para uno de los “skins” de un PJ.

4.6.1. Ejemplo práctico. Definición de la animación para uno de los “skins” de un PJ.

Para el ejemplo, utilizaremos un conjunto de sprites de los que se muestra un pequeño ejemplo a continuación:



Ilustración 24: Sprites utilizados para ejemplificar la creación de animaciones

Establecimiento de los estados que se utilizarán

Comenzaremos definiendo los estados. Para simplificar, nos limitaremos a los estados típicos:

- REPOSO: El personaje se mantiene en la misma posición, realizando una acción simple (como pueda ser saludar o respirar. En el ejemplo, dará vueltas sobre su eje)
- CAMINANDO_ESTE: El personaje se desplaza hacia la derecha.
- CAMINANDO_OESTE: El personaje se desplaza hacia la izquierda
- CAMINANDO_ARRIBA: El personaje “sube” por la pantalla
- CAMINANDO_ABAJO: El personaje “baja” por la pantalla.

Obtención de las imágenes

Hemos adquirido un conjunto de unas 19 imágenes. Supongamos que se encuentran en una carpeta llamada “imágenes”, relativa a la ruta de la AGM. Adjuntaremos las correspondientes a la animación en la que el personaje da vueltas sobre su propio eje.



Ilustración 25: Sprites para la animación del personaje dando vueltas

La ruta de cada una de estas imágenes será entonces:
imágenes\spinX.gif, donde X es un número de 1 a 7.

Asociación de imágenes a estados

Ahora, corresponderá asociar a cada estado el conjunto de imágenes correcto, indicando además parámetros para cada imagen. Ejemplificaremos el proceso para el estado “CAMINANDO_OESTE”

Nº de imágenes: 3

Imagen 1:

Ruta: imágenes\walk10.gif

Incremento en X: -3 píxels

Incremento en Y: 0 píxels.

Durante cuántos marcos se utilizará esta imagen: 2

Imagen2:

Ruta: imágenes\walk11.gif

Incremento en X: -3 píxels

Incremento en Y: 0 píxels

Número de marcos: 2

Imagen3:

Ruta: imágenes\walk12.gif

Incremento en X: -3 píxels

Incremento en Y: 0 píxels

Número de marcos: 2

Procederíamos análogamente para los restantes estados.

Determinación del área permitida

Asumiremos que el PJ responderá a pulsaciones del ratón sobre la pantalla, dirigiéndose al punto señalado. Limitaremos el área por la que puede moverse a un rectángulo como mostraremos en la figura (sombreada):

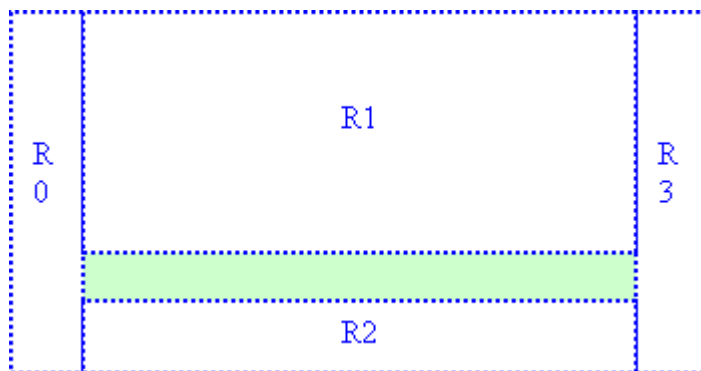


Ilustración 26 Ejemplo de definición de área

Nótese cómo se han definido los rectángulos (para más información acerca de las restricciones a la hora de delimitar la región permitida, consúltese [2.1.6](#) y [2.2.2.7](#))

Si la pantalla tiene un tamaño de 800x600, y el área permitida está constituida por el rectángulo que tiene su esquina superior izquierda en (25,450) con un ancho de 750 y un alto de 125 píxels, el área se establecerá como sigue:

Número de rectángulos: 4

Rectángulo 0:

X:0
Y:0
ANCHO:25
ALTO:600

Rectángulo 1:

X:25
Y:0
ANCHO:750
ALTO:450

Rectángulo 2:

X:25
Y: 575 (450+125)
ANCHO:750
ALTO:25 (600-575)

Rectángulo 3:

X:775 (25+750)
Y:0
ANCHO:25 (800-25)
ALTO:600

Creación del XML

Ahora que ya tenemos toda la información precisa, podremos crear el fichero de configuración.


```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE ANIMACION SYSTEM "xml/anim5.dtd" >
<ANIMACION>

<!-- lista de im&aacute;genes -->

<LIMAGENES tamanho="19">
<IMAGEN>imagenes/spin1.gif</IMAGEN>
<IMAGEN>imagenes/spin2.gif</IMAGEN>
<IMAGEN>imagenes/spin3.gif</IMAGEN>
<IMAGEN>imagenes/spin4.gif</IMAGEN>
<IMAGEN>imagenes/spin5.gif</IMAGEN>
<IMAGEN>imagenes/spin6.gif</IMAGEN>
<IMAGEN>imagenes/spin7.gif</IMAGEN>
<IMAGEN>imagenes/walkf0.gif</IMAGEN>
<IMAGEN>imagenes/walkf1.gif </IMAGEN>
<IMAGEN>imagenes/walkf2.gif</IMAGEN>
<IMAGEN>imagenes/walkl0.gif</IMAGEN>
<IMAGEN>imagenes/walkl1.gif</IMAGEN>
<IMAGEN>imagenes/walkl2.gif</IMAGEN>
<IMAGEN>imagenes/walkr0.gif</IMAGEN>
<IMAGEN>imagenes/walkr1.gif</IMAGEN>
<IMAGEN>imagenes/walkr2.gif</IMAGEN>
<IMAGEN>imagenes/walkb0.gif</IMAGEN>
<IMAGEN>imagenes/walkb1.gif</IMAGEN>
<IMAGEN>imagenes/walkb2.gif</IMAGEN>
</LIMAGENES>

<!-- lista de estados, con sus correspondientes patrones de movimiento -->

<LESTADOS tamanho="5">
<ESTADO nombre="REPOSO" tipo="patron">
<PATRON>
    <ENTRADA>
        <INCX>0</INCX>
        <INCY>0</INCY>
        <MOVS>3</MOVS>
        <IDIMG>0</IDIMG>
        <!-- idimg es el índice de 0 a n-1 de la lista de imágenes -->
        <IDACC>0</IDACC>
    </ENTRADA>
    <ENTRADA>
        <INCX>0</INCX>
        <INCY>0</INCY>
        <MOVS>3</MOVS>
        <IDIMG>1</IDIMG>
        <IDACC>0</IDACC>
    </ENTRADA>
    <ENTRADA>
        <INCX>0</INCX>
        <INCY>0</INCY>
        <MOVS>3</MOVS>
        <IDIMG>2</IDIMG>
        <IDACC>0</IDACC>
    </ENTRADA>
    <ENTRADA>
        <INCX>0</INCX>
        <INCY>0</INCY>
        <MOVS>3</MOVS>

```

```

        <IDIMG>3</IDIMG>
        <IDACC>0</IDACC>
    </ENTRADA>
</ENTRADA>
    <ENTRADA>
        <INCX>0</INCX>
        <INCY>0</INCY>
        <MOVS>3</MOVS>
        <IDIMG>4</IDIMG>
        <IDACC>0</IDACC>
    </ENTRADA>
</ENTRADA>
    <ENTRADA>
        <INCX>0</INCX>
        <INCY>0</INCY>
        <MOVS>3</MOVS>
        <IDIMG>5</IDIMG>
        <IDACC>0</IDACC>
    </ENTRADA>
</ENTRADA>
    <ENTRADA>
        <INCX>0</INCX>
        <INCY>0</INCY>
        <MOVS>3</MOVS>
        <IDIMG>6</IDIMG>
        <IDACC>0</IDACC>
    </ENTRADA>
</PATRON>
</ESTADO>
<ESTADO nombre="caminando_este" tipo="patron">
<PATRON>
    <ENTRADA>
        <INCX>3</INCX>
        <INCY>0</INCY>
        <MOVS>2</MOVS>
        <IDIMG>13</IDIMG>
        <IDACC>0</IDACC>
    </ENTRADA>
    <ENTRADA>
        <INCX>3</INCX>
        <INCY>0</INCY>
        <MOVS>2</MOVS>
        <IDIMG>14</IDIMG>
        <IDACC>0</IDACC>
    </ENTRADA>
    <ENTRADA>
        <INCX>3</INCX>
        <INCY>0</INCY>
        <MOVS>2</MOVS>
        <IDIMG>15</IDIMG>
        <IDACC>0</IDACC>
    </ENTRADA>
</PATRON>
</ESTADO>

<!-- ojo aqu&iacute;, que en JAVA las Y positivas van hacia abajo. -->

<ESTADO nombre="caminando_norte" tipo="patron">
<PATRON>
    <ENTRADA>
        <INCX>0</INCX>

```

```

        <INCY>-3</INCY>
        <MOVS>2</MOVS>
        <IDIMG>16</IDIMG>
        <IDACC>0</IDACC>
    </ENTRADA>
    <ENTRADA>
        <INCX>0</INCX>
        <INCY>-3</INCY>
        <MOVS>2</MOVS>
        <IDIMG>17</IDIMG>
        <IDACC>0</IDACC>
    </ENTRADA>
    <ENTRADA>
        <INCX>0</INCX>
        <INCY>-3</INCY>
        <MOVS>2</MOVS>
        <IDIMG>18</IDIMG>
        <IDACC>0</IDACC>
    </ENTRADA>
</PATRON>
</ESTADO>
<!-- ..... -->
<ESTADO nombre="caminando_sur" tipo="patron">
<PATRON>
    <ENTRADA>
        <INCX>0</INCX>
        <INCY>3</INCY>
        <MOVS>2</MOVS>
        <IDIMG>7</IDIMG>
        <IDACC>0</IDACC>
    </ENTRADA>
    <ENTRADA>
        <INCX>0</INCX>
        <INCY>3</INCY>
        <MOVS>2</MOVS>
        <IDIMG>8</IDIMG>
        <IDACC>0</IDACC>
    </ENTRADA>
    <ENTRADA>
        <INCX>0</INCX>
        <INCY>3</INCY>
        <MOVS>2</MOVS>
        <IDIMG>9</IDIMG>
        <IDACC>0</IDACC>
    </ENTRADA>
</PATRON>
</ESTADO>
<!-- ..... -->
<ESTADO nombre="caminando_oeste" tipo="patron">
<PATRON>
    <ENTRADA>
        <INCX>-3</INCX>
        <INCY>0</INCY>
        <MOVS>2</MOVS>
        <IDIMG>10</IDIMG>
        <IDACC>0</IDACC>
    </ENTRADA>
    <ENTRADA>
        <INCX>-3</INCX>
        <INCY>0</INCY>
        <MOVS>2</MOVS>

```

```

        <IDIMG>11</IDIMG>
        <IDACC>0</IDACC>
    </ENTRADA>
    <ENTRADA>
        <INCX>-3</INCX>
        <INCY>0</INCY>
        <MOVS>2</MOVS>
        <IDIMG>12</IDIMG>
        <IDACC>0</IDACC>
    </ENTRADA>
</PATRON>
</ESTADO>
</LESTADOS>

<!-- se asume que la pantalla tiene una resolucion 800x600 -->
<AREA tamanho="4">
    <RECTANGULO>
        <X>0</X>
        <Y>0</Y>
        <ANCHO>25</ANCHO>
        <ALTO>600</ALTO>
    </RECTANGULO>

    <RECTANGULO>
        <X>25</X>
        <Y>0</Y>
        <ANCHO>750</ANCHO>
        <ALTO>450</ALTO>
    </RECTANGULO>

    <RECTANGULO>
        <X>25</X>
        <Y>575</Y>
        <ANCHO>750</ANCHO>
        <ALTO>25</ALTO>
    </RECTANGULO>

    <RECTANGULO>
        <X>775</X>
        <Y>0</Y>
        <ANCHO>25</ANCHO>
        <ALTO>600</ALTO>
    </RECTANGULO>
</AREA>
</ANIMACION>

```

Adición de nuevos comportamientos

No añadiremos más posibilidades a las capacidades del PJ. Por tanto, omitiremos esta parte. Para obtener ayuda acerca de cómo hacer esto, consúltase el programa de pruebas y la parte integrada (sería añadir nuevos métodos del estilo “moverse(X,Y)”, a grandes rasgos)

4.7. Fuentes de sprites

Todos los sprites utilizados durante las pruebas se descargaron de internet. Algunas direcciones útiles para obtener más sprites son:

- <http://www.gsarchives.net/index2.php>
- <http://rpg-icons.com/>
- <http://www.sonicworld.net/sws/sprites/sonic/>
- <http://www.panelmonkey.org/category.php?id=1>
- <http://tsgk.captainn.net/?d=text&s=GameBoy%20Advance>

Sin embargo, la mayor parte de estos sprites han sido extraídos de juegos comerciales, por lo que no se recomienda su uso en sistemas finales.

5. Troubleshooting

5.1. *Objetos en el inventario inicial*

Descripción: Para algunas historias podría ser interesante que el inventario con el cual es creado un personaje no se encontrara a vacío si no que tuviera por defecto algunos objetos que le sirvieran para empezar sus pesquisas.

Se encontró que la implementación de meter en el inventario dichos objetos cuando se creaba el jugador por medio de una inserción “manual” en su inventario no era fiable ya que esto no era recogido en la base de datos y por lo tanto al iniciar la partida con el jugador creado al intentar refrescar su información con la de la base de datos se perdían dichos objetos.

Solución: Siendo KernelBean.java quien creaba el ClienteJugador se aprovecho que la llamada a creaPersonajeJ() solo se hacia en el caso de un nuevo jugador (no para cada vez que se conecta) de esta manera se definió una nueva función objetosIniciales() para que devolviera un inventario inicializado con los objetos por defectos con el que CreaPersonajeJ() crea a dicho personaje nuevo.

5.2. *Cierre anómalo del cliente con el botón aspa*

Descripción: Se observo que cuando se intentaba cerrar el cliente por medio del botón aspa (esquina superior derecha) en algunas ocasiones se denegaba la acción alegando que el jugador tenia una conversación pendiente. La razón era que existe un atributo llamado idPNJBloqueante que alberga en el cliente el id del personaje con el que esta hablando y cuando el cliente intenta cerrar se requiere que este id este a null. El problema era que este atributo debía dársele valor solo cuando se ejecutase con éxito la acción hablar con un personaje no jugador sin embargo las condiciones que controlaban esto eran erróneas y confundían este caso con el de cruzar una puerta, por ejemplo, de manera que el atributo podía dejar de ser null sin necesidad de que se hubiera iniciado una conversación.

Solución: Se traslado la decisión de darle valor a idPNJBloqueante al momento en el que se mostraba la primera frase de la conversación, de manera que se aseguraba tanto que la acción solicitada era la de hablar como de que no había sido rechazada pro el personaje no jugador. Una vez realizado esto no se volvió a reproducir el problema.

5.3. *Al ejecutar la acción USAR de un objeto no se recibe respuesta y se pierde la interacción con dicho objeto*

Descripción: Si al ejecutar la accion USAR de un objeto sobre otro no se obtienen ningún tipo de respuesta y luego se observa que ya ninguna otra accion da algún resultado sobre dicho objeto es probable que los pasos previos a devolver la accion para ejecutar USAR estén mal implementados en las líneas en que se obtiene la habitación del objeto sobre el que se va a usar el 1º objeto dándose el caso de que devuelve null en vez de una habitación, lo cual hace que cuando se intenta buscar en dicha habitación el 2º objeto salte una excepción que provoca como efecto secundario la perdida del primer objeto.

Solución: Generalmente al búsqueda del objeto sobre el que se va a usar el primero se hace con la instrucción *Habitacion h = Acciones.conseguirHabitacion(idHab);* pero esto solo es valido para objetos encontrados en una habitación, es decir, los objetos que

están en el inventario no tendrán un idHab asociado a ninguna habitación, así que la búsqueda sería errónea, para este caso. Teniendo en cuenta que un objeto del inventario esta donde este el jugador basta con que sustituyamos esa línea por unas donde se busque la habitación donde esta el jugador: *HabitacionHome habHome = Acciones.getHabitacionHome(); Habitacion h = habHome.findByIdPJ(idPJ);*
Se ha incluido a posteriori en la plantilla el siguiente código para podernos olvidar de tal consideración:

```
If (idHab!=null) { Habitacion h = Acciones.conseguirHabitacion(idHab); }  
else { HabitacionHome habHome = Acciones.getHabitacionHome(); Habitacion h =  
habHome.findByIdPJ(idPJ);}
```

5.4. El servidor jboss no consigue conectar y cae en un sin fin de intentos de reconexión

Descripción: Al arrancar el servidor jboss se observa que cuando esta a punto de conectar salta una excepción para finalmente intentar reconectar obteniendo el mismo resultado una y otra vez.

Solución: El motivo puede ser que la tabla JMS_MESSAGES se encuentre saturada de mensajes, en tal caso bastara con borrar su contenido manualmente. Para ello diríjase al servicio Hypersonic del servidor (<http://localhost:8080/jmx-console/>) y ejecute la operación startDatabaseManager() desde la cual ejecutar la sentencia SQL *DELETE FROM JMS_MESSAGES*

5.5. Algunas acciones se repiten sin ninguna razón o patrón aparente

Descripción: Puede darse el caso de que se vea en momentos de transición, como al entrar en una nueva habitación o al desconectar o cuando entra un nuevo usuario, que se repita alguna operación (generalmente la ultima realizada).

Solución: Esto se debe a que el mensaje que desencadena dicha operación no ha sido borrado de la cola (tabla JMS_MESSAGES) de manera que se ha vuelto a desencadenar su ejecución. Para evitarlo hay que cerciorarse de que cuando se emita un mensaje se le de un tiempo de vida que asegure su eliminación tras la primera y única ejecución. Se trata de que siendo *publisher* el *TopicPublisher* que emite un mensaje *m* se haga de la siguiente manera: *publisher.publish(m, DeliveryMode.NON_PERSISTENT, Message.DEFAULT_PRIORITY, 1000);*

Para completa se incluye esta configuración en el jboss modificando el archivo de configuración *server\default\conf\standardjboss.xml* en el nodo *invoker-proxy-binding* (de *invoker-proxy-bindings*) con atributo name *message-driven-bean* poniendo su parámetro *TimeToLive* a 1000 (por defecto vendrá a 0 que simboliza infinito).

5.6. Al realizar una conversación algunas frases no aparecen

Descripción: Este fallo se observa cuando en una conversación en la que se deberían suceder varias frases entre un personaje jugador y uno no jugador algunas de estas, generalmente las del jugador, no aparecen. Esto siempre referido a la parte fija de la conversación.

Solución: El problema se debe a que el muestreo de estas frases en la interface grafica ha de realizarse de manera sincronizada para que las frases del personaje no jugador no pisen a las del jugador (o viceversa). Esto se soluciona envolviendo en EscritorTexto.java las partes en que se mostraban y retiraban las frases en métodos synchronized.

5.7. Desaparecen los objetos de un inventario

Descripción: Puede verse que al volver a conectarse con algún jugador tengamos el inventario vacío a pesar de que en la anterior partida habíamos recolectado una serie de objetos

Solución: Hay que tener en cuenta que al hacer modificaciones en las clases del servidor después hay que volver a desplegar y esto tiene como efecto secundaria que tablas que afectan al inventario se queden vacías. Esto en principio no tiene solución

5.8. Inclusión de librerías

Descripción: Para solucionar el siguiente error, que aparece como en la consola del Eclipse en la zona de errores y advertencias:

```
<< The type javax.jms.MessageListener cannot be resolved. Its indirectly referenced from required .class files.>>
```

Solución: La solución radica en incluir las librerías del JBoss, que se encuentran en la carpeta lib, en este caso concreto, se trata de incluir la librería jboss-jmx.jar .

5.9. Solución a error del JBoss

Descripción: Para solucionar la siguiente excepción, que impide que el servidor arranque:

```
javax.jms.JMSEException: Error creating the dlq connection: XAConnectionFactory not bound
at org.jboss.ejb.plugins.jms.DLQHandler.createService(DLQHandler.java:152)
at org.jboss.system.ServiceMBeanSupport.create(ServiceMBeanSupport.java:158)
at org.jboss.ejb.plugins.jms.JMSContainerInvoker.innerCreate(JMSContainerInvoker.java:394)
at org.jboss.ejb.plugins.jms.JMSContainerInvoker.startService(JMSContainerInvoker.java:579)
at
org.jboss.ejb.plugins.jms.JMSContainerInvoker$ExceptionHandlerImpl.onException(JMSContainerInvoker.java:1079)
at org.jboss.ejb.plugins.jms.JMSContainerInvoker$1.run(JMSContainerInvoker.java:591)
11:27:17,328 INFO [JMSContainerInvoker] Trying to reconnect to JMS provider
```

Solución: Se debe comprobar en el fichero de configuración de jboss (jboss-service.xml) que la siguiente línea aparece tal y como se muestra a continuación:

```
<attribute name="RecursiveSearch">True</attribute>
```

Y no con dicho atributo a False.

5.10. Carga de la ip

Descripción: Si en algún momento se ve aparecer el siguiente mensaje por consola:

```
<Error al cargar la ip>
```

Significará que el fichero ip.xml no se encuentra situado en el lugar que debería ocupar para que la aplicación haga un uso correcto y adecuado del mismo.

Solución: Hay que ubicarlo en el lugar que le corresponde, es decir en el directorio de trabajo actual.

5.11. Conexión al juego

Descripción: Es probable que en alguna ocasión aparezca un panel informativo informando sobre la imposibilidad de efectuar una conexión con un personaje determinado. Más o menos el mensaje viene a ser el que se muestra a continuación:

```
<< Se te ha denegado la conexión. Causa: Tu id y password no se corresponden con ninguno de los jugadores desconectados >>
```

Solución: Esto se debe a que el personaje no está creado y estás intentando conectarlo, puede deberse a que no exista la adecuada sincronización entre las acciones de creación y sincronización o puede deberse a que estés intentando conectar a un personaje ya creado pero que hay permanece activo en el juego, es decir, que no se ha desconectado correctamente por alguna causa, bien fallo de la aplicación o error del usuario.

5.12. Null Pointer Exception

Descripción: Si aparece una Null Pointer Exception que incluye información similar a la que a continuación se muestra:

```
at org.jboss.mq.server.JMSTopic.receive(JMSTopic.java:260)
at org.jboss.mq.server.ClientConsumer.receive(ClientConsumer.java:225)
```

Solución: No hay que concederle mayor importancia, puesto que es algo habitual, que ocurre también al ejecutar distintas acciones durante una partida normal dentro del AGM.

5.13. Puerto ocupado

Descripción: Si en algún momento aparece alguna excepción informando de que el puerto por que se comunica el servidor JBoss, esto es el 8080, está ocupado por alguna otra aplicación.

Solución: Debemos comprobar si existen más instancias activas de Eclipse mediante el comando `ps` y sí se verifica este punto hay que proceder a eliminar dichas instancias empleando para ello la orden `killall -9 java`.

5.14. Out of memory

Descripción: El siguiente error es un error del que no se ha encontrado el origen y por tanto tampoco la solución al mismo:

```
java.rmi.ServerError: Error occurred in server thread; nested exception is:
java.lang.OutOfMemoryError
at sun.rmi.server.UnicastServerRef.dispatch(UnicastServerRef.java:289)
at sun.rmi.transport.Transport$1.run(Transport.java:148)
at java.security.AccessController.doPrivileged(Native Method)
at sun.rmi.transport.Transport.serviceCall(Transport.java:144)
at sun.rmi.transport.tcp.TCPTransport.handleMessages(TCPTransport.java:460)
at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run(TCPTransport.java:701)
at java.lang.Thread.run(Thread.java:534)
at sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer(StreamRemoteCall.java:247)
at sun.rmi.transport.StreamRemoteCall.executeCall(StreamRemoteCall.java:223)
at sun.rmi.server.UnicastRef.invoke(UnicastRef.java:133)
at org.jboss.invocation.jrmp.server.JRMPInvokerStub.invoke(Unknown Source)
at org.jboss.invocation.jrmp.interfaces.JRMPInvokerProxy.invoke(JRMPInvokerProxy.java:135)
at org.jboss.invocation.InvokerInterceptor.invoke(InvokerInterceptor.java:87)
at org.jboss.proxy.TransactionInterceptor.invoke(TransactionInterceptor.java:46)
at org.jboss.proxy.SecurityInterceptor.invoke(SecurityInterceptor.java:45)
at org.jboss.proxy.ejb.HomeInterceptor.invoke(HomeInterceptor.java:173)
at org.jboss.proxy.ClientContainer.invoke(ClientContainer.java:85)
```

```
at $Proxy0.create(Unknown Source)
at agm.clienteAdministrador.ClienteAdministrador.<init>(ClienteAdministrador.java:70)
at pruebasxml.PruebaCliente.<init>(PruebaCliente.java:71)
at pruebasxml.PruebasParser.main(PruebasParser.java:31)
Caused by: java.lang.OutOfMemoryError
```

Solución: Como no es un error que ocurra siempre, ni tan siquiera muy a menudo y solo se ha producido en ordenadores privados y no en el servidor ingenias es probable que se deba a una configuración hardware concreta y no se vuelva a presentar.

5.15. Error en carga de imagenes

Descripción: Error en la carga de las imagenes, en el programa de pruebas. Se lanzaban NullPointerException

Solución: La carpeta de imagenes había sido borrada accidentalmente. Basto con volver a incluirla, para que el error remitiese

5.16. Error puntual en lectura de XML

Descripción: Excepción MalformedProtocol, durante la carga de ficheros XML de configuración de animaciones. Solo se produjo en una maquina concreta, durante el inicio de la fase de integración de la animación en la aplicación.

Solución: Probablemente se tratara de un error con las librerías de Xerces, debido a una versión diferente a la que se había estado utilizando. Al cambiar de maquina, el error no ha vuelto a reproducirse.

5.17. Excepción debida a la ordenación del código

Descripción: NullPointerExceptions durante la depuración del paquete de animaciones integrado con AGM.

Solución: Reordenación del código de inicialización de la interfaz, y de los objetos existentes en pantalla en ese momento. La razón obedece a que ambos elementos se comunican, y debido al orden previo, se producían pasos de argumentos con referencias a objetos todavía no inicializados.

6. Bibliografía

Brackeen, Daniel

Developing Games in Java

Indianapolis, New Riders, copyright © 2004

“Professional EJB” Rahim Adatia, Faiz Arni, Craig A. Berry, Kyle Gabhart, John Griffin.

Work Press Ltd

Tusc tutorial: <http://www.tusc.com.au/tutorial/html/>

TUTORIAL DE LOMBOZ: <http://www.objectlearn.com/support/docs/index.jsp>

XDOCLET DOCUMENTATION: <http://xdoclet.sourceforge.net/olddocs/>

JBOSS, Eclipse and Lomboz.

<http://www.jboss.org/index.html?module=bb&op=viewtopic&t=36910>

http://www.fdi.ucm.es/profesor/jjgomez/lp3_2003_2004/

<http://www.gsarchives.net/index2.php>

<http://rpg-icons.com/>

<http://www.sonicworld.net/sws/sprites/sonic/>

<http://www.panelmonkey.org/category.php?id=1>

<http://tsgk.captainn.net/?d=text&s=GameBoy%20Advance>

7. Palabras clave

J2SE

J2EE

EJB

Bean

Eclipse

Lomboz

Jboss

Aventura gráfica Multi-jugador

XML

Sprite

8. Cesión de derechos

El equipo de este proyecto (Tania Anta, Nazaret Sánchez, Santos Herranz) autorizan a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado

Tania Anta Álvarez

Nazaret Sánchez Rodríguez

Santos Hernanz Lillo

Madrid a 30 de Junio de 2005

