UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA
Departamento de Ingeniería del Software e Inteligencia Artificial

ARQUITECTURA Y METODOLOGÍA PARA EL DESARROLLO DE SISTEMAS EDUCATIVOS BASADOS EN VIDEOJUEGOS

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Marco Antonio Gómez Martín

Bajo la dirección del doctor
Pedro Antonio González Calero

Madrid, 2007

- **ISBN: 978-84-692-0154-1**

Arquitectura y metodología para el desarrollo de sistemas educativos basados en videojuegos

TESIS DOCTORAL

Marco Antonio Gómez Martín

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Octubre 2007

Arquitectura y metodología para el desarrollo de sistemas educativos basados en videojuegos

Memoria que presenta para optar al título de Doctor en Informática

Marco Antonio Gómez Martín

Dirigida por el Doctor

Pedro Antonio González Calero

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Octubre 2007

A mis padres

En todo hombre hay escondido
un niño que quiere jugar
Friedrich W. Nietzsche (1844-1900)

Agradecimientos

A todos los que la presente vieren y entendieren.
Inicio de las Leyes Orgánicas. Juan Carlos I

Tengo que reconocer que cuando, allá por mi primer año en la Facultad, conocí al que a la postre se convertiría en mi director de Tesis, no me gustó nada. La primera impresión que me causó fue bastante mala, como le descubrí un día mucho tiempo después. La casualidad hizo que al año siguiente volviera a formar parte del grupo de profesores que trataban de llenarnos la cabeza de bits, puertas lógicas, lenguajes de programación y formalismos. Y entonces mi concepto sobre él dio un vuelco. Pedro no era, ni mucho menos, una persona desagradable.

Al terminar la carrera inicié, convencido por la inigualable Carmen Fernández, los estudio de Doctorado en el extinto Departamento de Sistemas Informáticos y Programación. Después del primer año, cuando en Septiembre tenía que decidirme sobre en qué campo concreto investigar, tenía las ideas bastante poco claras; no sabía quién podría ser mi tutor en el trabajo del segundo año ni qué tema elegir. Y entonces, cuidando un examen en el aula magna de físicas, mirando hacia los alumnos junto a Pedro, profesor de aquella asignatura, éste me miró y me dijo "he estado pensando este verano en vosotros". Se refería a mi hermano y a mí. Mientras los alumnos trataban de sacar adelante el examen, me propuso una serie de temas de investigación que no pude (pudimos) rechazar. Por eso, y por todo lo que vino después, por haber creído en mí cuando yo no lo hacía, por el apoyo durante todo este tiempo, mi primer agradecimiento es para él.

Dentro del departamento hay otro montón de personas que me han ayudado, apoyado o contado conmigo durante este tiempo, y a los que también tengo mucho que agradecer. A la primera ya la he nombrado, Carmen Fernández, que cuando aún estaba en quinto y nos cruzábamos por los pasillos nos preguntaba por nuestras ideas para el futuro y tuvo la amabilidad de reservar una mañana para explicarnos las posibilidades que ofrecía la Universidad. Belén Díaz, con su empeño en ver fáciles las cosas difíciles, también ha potenciado aspectos de este trabajo que yo creía muros insalvables. Guillermo Jiménez, ávido lector de artículos, siempre podía relacionar mis ideas

con las de algún otro investigador. Luis Hernández, magnífico organizador, siempre interesado por mis avances y dispuesto a ayudar en todo. Y Juan Pavón y Jorge Gómez que me acogieron en su grupo de investigación durante un año. En general, gracias a todos los miembros del grupo de investigación de Aplicaciones de Inteligencia Artificial en el que he desarrollado el trabajo, y a los miembros del Departamento de Ingeniería del Software e Inteligencia Artificial; de las conversaciones con vosotros durante la comida es difícil no aprender algo. Finalmente, tampoco puedo olvidar a los becarios y estudiantes que han contribuido en la implementación de JV^{2}M, como Pablo Palmier que me demostró que los becarios responsables existen.

Pero sin duda la persona más importante ha sido Pedro Pablo. Me resulta muy difícil encontrar palabras de agradecimiento para él por varios motivos. El primero es que ha sido desde siempre un pilar indispensable para mí; el segundo es la imposibilidad de agradecerle el haber aguantado mis ideas más estrambóticas y haber sido crítico y revisor de todas las esquinas de esta tesis; y el último es que, si encontrara las palabras de agradecimiento y las expresara en alto, me diría que ya las conocía.

También tengo que agradecer a aquellos que durante este tiempo me han apoyado y han soportado mis ausencias. Y eso va *especialmente* por Laura, que ha competido por mi tiempo con esta criatura que tienes entre tus manos, y casi nunca ha salido ganando. Gracias.

Gracias al resto de mi familia. A mi hermano Jesús Javier, *Joti*; a mis abuelos, por la ropa de punto, las paellas y los paseos por el retiro; y por supuesto a mis tíos. A Benjamín por inculcarme la curiosidad por el conocimiento por el mero hecho de saber, por las leyes de Kepler y por el método para calcular eclipses que nunca llegó. A Julio, por explicar entre sudores a ese niño de seis años que una vez fui qué era el parlamento. A Alfredo, que nos traía a casa la ilusión en piezas de un nuevo ordenador, y lo montaba con dos moscones alrededor deseosos de absorber todo lo que veían. A Marisa que, con Benja, me ayudaron a refinar el ejemplo de animales que aparece en el capítulo 4 mientras se extrañaba de que les preguntara interesado si todos los animales que vuelan y tienen plumas son pájaros. Y a Quique, por las entretenidas conversaciones sobre los últimos recovecos de C++, sobre librerías de enlace dinámico, y sobre cualquier otra cosa que se precie.

Y para terminar dejo para el último lugar a los primeros: a mis padres. Aquí la tenéis. Esta tesis es vuestra.

Resumen

Desocupado lector, sin juramento me podrás creer que quisiera que este libro [...] fuera el más hermoso, el más gallardo y más discreto que pudiera imaginarse.

Miguel de Cervantes, Don Quijote de la Mancha

Este trabajo de Tesis Doctoral presenta una arquitectura y metodología de desarrollo de sistemas educativos basados en videojuegos. El objetivo principal es:

1. Minimizar la dependencia entre el conocimiento específico del dominio que se enseña y el resto del sistema. De esta forma, el conocimiento del dominio puede aprovecharse para implementar otros videojuegos educativos que enseñen lo mismo pero de distinta forma. También permite utilizar las partes no específicas del dominio en varios juegos educativos. En definitiva, permite la *reutilización*, de forma que se reduzcan los costes de creación de este tipo de aplicaciones.

2. Permitir que en desarrollos posteriores sean fácilmente sustituibles algunos módulos. Principalmente se persigue poder *intercambiar* fácilmente aquellos módulos susceptibles de quedar obsoletos con el avance tecnológico.

3. Permitir que los distintos profesionales que entran en juego durante el periodo de desarrollo puedan trabajar sin solaparse unos con otros, minimizando las dependencias entre ellos.

La metodología propuesta aboga por una clara división entre los contenidos pedagógicos y los contenidos lúdicos que los rodean. Para poderla hacer efectiva, la arquitectura software subyacente divide la aplicación en módulos con unas responsabilidades claras y no solapadas.

Como prueba del correcto funcionamiento de la metodología y arquitectura aquí propuesta, el trabajo incluye la descripción detallada del desarrollo de JV^2M, un sistema educativo para enseñar la compilación de un lenguaje de alto nivel orientado a objetos como es Java.

Índice

Índice de figuras

Índice de Tablas

Capítulo 1

Introducción

Púsose don Quijote delante de dicho carro, y haciendo en su fantasía uno de los más desvariados discursos que jamás había hecho, dijo en alta voz:

Alonso Fernández de Avellaneda, El Ingenioso Hidalgo Don Quijote de la Mancha

1.1. Motivación

Varias décadas de investigación en aprendizaje por ordenador han demostrado que los expertos del dominio deben involucrarse en el proceso de creación de los contenidos de las aplicaciones educativas para que el resultado sea satisfactorio. El resultado ha sido un desmesurado esfuerzo en la construcción de diversas herramientas de autoría, que permiten directamente a esos expertos la construcción de estas aplicaciones.

Durante años una inmensa colección de programas educativos han visto la luz, gracias a estas herramientas que permiten generar contenidos en formatos determinados que luego son presentados por *visores* genéricos. Desgraciadamente, en la mayoría de los casos la *interacción* con el usuario es muy limitada, haciendo que la motivación inicial por el aprendizaje se diluya en la constante visión de contenidos poco interactivos.

Para mejorar la motivación, los últimos años han sido testigos del surgimiento de un nuevo tipo de aprendizaje por computadora utilizando videojuegos, en lo que se ha venido a llamar aprendizaje basado en juegos (*game-based learning*) o "juegos serios" (*serious games*) (Gómez Martín, Gómez Martín y González Calero, 2004a).

Resulta fácil defender el argumento de que utilizar videojuegos para enseñar es efectivo. Al fin y al cabo, los juegos se han utilizado desde siempre para enseñar, e incluso los cachorros de muchas especies lo utilizan como método de entrenamiento en un entorno seguro.

Sin embargo, en el momento de desarrollar una de estas aplicaciones educativas, nos enfrentamos a muchas más dificultades de las que teníamos con las aplicaciones educativas tradicionales.

Desde el punto de vista del *contenido*, en los juegos tradicionales éste toma dos formas distintas: multimedia y jugabilidad. Los modelos tridimensionales de los escenarios, objetos y personajes, las texturas bidimensionales que los *colorean*, las animaciones, músicas y efectos sonoros definen el contenido multimedia, mientras que la jugabilidad viene determinada por "lo que el usuario hace" dentro del juego. Los diseñadores encargados de la jugabilidad construyen descripciones detalladas de las acciones que el jugador puede hacer y cómo el juego debe reaccionar.

Cuando añadimos la capacidad educativa a uno de estos juegos, debemos sumar a estos dos tipos de contenidos el del dominio que se pretende enseñar. En este caso, el experto del dominio debe *también* proporcionar descripciones detalladas de lo que el estudiante debe hacer y cómo debe reaccionar el sistema, tarea que se solapa por completo con la del diseñador del juego.

Por si eso fuera poco, el tamaño y calidad de los videojuegos no deja de incrementar, hasta el punto de que para poder satisfacer las expectativas de los ávidos jugadores, hoy por hoy se puede llegan a necesitar varios cientos de personas trabajando durante al menos dos años.

Debido a todo esto, parece claro que la construcción de videojuegos educativos es mucho más complicada que la creación de aplicaciones educativas tradicionales, lo que provoca que el presupuesto necesario difiera en varios órdenes de magnitud. La consecuencia directa de estas dificultades es que hoy por hoy no existen demasiados estudios empíricos que demuestren que efectivamente su uso es más efectivo que las aplicaciones educativas tradicionales.

Para aliviar el problema, este trabajo de Tesis Doctoral presenta una arquitectura y metodología de desarrollo de sistemas educativos basados en videojuegos. La arquitectura dirigida por datos permite la construcción de videojuegos educativos permitiendo la creación por separado de los contenidos de instrucción y los de jugabilidad. Gracias a esta división, el solapamiento en las tareas de ambos expertos se reduce al mínimo. La metodología también propone una serie de herramientas para comprobar que los contenidos generados por ambos expertos se adecúan al producto final perseguido.

El trabajo ejemplifica además esa arquitectura con un sistema de enseñanza llamado JV^{2}M, para enseñar la compilación de un lenguaje de alto nivel orientado a objetos como es Java.

1.2. Estructura de capítulos

A continuación aparece la secuencia de capítulos, así como una breve descripción de su contenido:

Capítulo 2: Arquitecturas de videojuegos. Este capítulo introduce las partes más importantes en las que se estructura una aplicación de entretenimiento. En particular, describiremos la importancia de una arquitectura basada en datos, justificaremos la necesidad de la utilización de código de terceros y detallaremos cómo gestionar la ejecución de todas las tareas que debe realizar la aplicación. El capítulo pasa después a explicar algunas formas de manejar las entidades que forman parte del entorno virtual, y los lenguajes de scripts como un método de sacar parte de su control a ficheros de datos.

Capítulo 3: Aplicaciones educativas. Este capítulo hace un análisis de las arquitecturas de los programas educativos que centran nuestro foco de atención. Para eso, comienza por analizar la construcción de sistemas de enseñanza inteligentes. Después se analiza cómo se integran agentes en entornos virtuales para después aplicar esa integración a los agentes pedagógicos.

Capítulo 4: Una metodología para la creación de contenidos dirigidos por datos. Presenta la metodología de desarrollo de las aplicaciones educativas basadas en videojuegos. Para eso, primero se hace una descripción de alto nivel de las fases que deben seguirse durante su creación y las herramientas de soporte necesarias. También describimos nuestra propuesta de metodología para la creación de los contenidos que presenta la aplicación. Por último, el capítulo describe dos técnicas de uso del análisis formal de conceptos para el análisis de los contenidos creados.

Capítulo 5: Arquitectura para el desarrollo de sistemas educativos basados en videojuegos. Se presenta la propuesta de arquitectura de aplicaciones educativas basadas en videojuegos. Describe primero los objetivos, para pasar a mostrar la visión general de la misma. Posteriormente analiza cada uno de los módulos en detalle. El capítulo termina con una evaluación.

Capítulo 6: Sistema educativo para la enseñanza de la máquina virtual de Java. Este capítulo describe las dos versiones de JV^2M, el sistema educativo que permite aprender la máquina virtual de Java y cómo el código Java es compilado para ella. El sistema ha sido creado utilizando la metodología y arquitectura descritas en los dos capítulos anteriores, por lo que éste se centra en cómo se han llevado a cabo las directrices descritas en ellos.

Capítulo 7: Conclusiones y trabajo futuro.

1.3. Contribuciones

Todos los capítulos de esta tesis, a excepción de éste de Introducción y del último de Conclusiones, terminan con una sección titulada "Notas bibliográficas". En esas secciones se ponen en contexto todos aquellos artículos escritos por el autor y por el director de este trabajo que han sido publicados en distintas revistas, congresos y *workshops*; en concreto, se relaciona su contenido con el del capítulo en cuestión.

Aunque aconsejamos la lectura de estas secciones, por describir el contenido de los artículos propios y ajenos, a continuación aparece una lista de algunas de las publicaciones a las que ha dado lugar este trabajo de Tesis Doctoral. Para una lista exhaustiva, remitimos al lector a las secciones anteriormente mencionadas y a las citas bibliográficas que aparecen a partir de la página 271.

- Uso de juegos para educación: en Gómez Martín, Gómez Martín y González Calero (2004a), Gómez Martín, Gómez Martín y González Calero (2004b) y González Calero, Gómez Martín y Gómez Martín (2006) describimos el uso de juegos para la educación.
- Metodología y arquitectura: una descripción preliminar de la metodología y arquitectura presentadas en esta memoria fue expuesta en Gómez Martín, Gómez Martín y González Calero (2003). La arquitectura final es presentada en Gómez Martín, Gómez Martín y González Calero (2007a) y Gómez Martín, Gómez Martín y González Calero (2007b) con distintos grados de detalle.
- Arquitectura en otros contextos: la arquitectura descrita también se ha utilizado para la implementación de VirPlay, como se explica brevemente en Jiménez Díaz, Gómez Albarrán, Gómez Martín y González Calero (2005c). También hemos descrito en Peinado Gil, Gómez Martín y Gómez Martín (2005) una variación de la misma para el desarrollo de aplicaciones con una fuerte componente narrativa.
- Análisis de los contenidos: gracias a la separación entre los dos tipos de contenido, pedagógico y lúdico, la metodología permite aplicar técnicas de análisis a ambos por separado. Díaz Agudo, Gómez Martín, Gómez Martín y González Calero (2005) describe una técnica para analizar el contenido pedagógico; por su parte, Gómez Martín, Gómez Martín, González Calero y Díaz Agudo (2006b) hace lo propio con los datos relativos al juego.
- Descripción de JV^{2}M: la aplicación educativa para enseñar la estructura interna de la máquina virtual de Java ha sido descrito en Gómez

Martín, Gómez Martín y González Calero (2006a) y Gómez Martín, Gómez Martín, González Calero y Palmier Campos (2007d).

- Desarrollo de la aplicación: en Gómez Martín, Gómez Martín y Jiménez Díaz (2007c) se describen las herramientas necesarias para el desarrollo de aplicaciones educativas basadas en videojuegos. Por su parte, Gómez Martín y Gómez Martín (2006) describe, entre otras cosas, las características y diferencias entre los dos motores gráficos utilizados en JV^{2}M, Nebula 1 y Nebula 2.

Capítulo 2

Arquitecturas de videojuegos

Si los arquitectos hiciesen edificios de la misma forma en que los programadores escriben programas, el primer pájaro carpintero que pasase por aquí destruiría la civilización

Gerald Weinberg

Este capítulo introduce las partes más importantes en las que se estructura una aplicación de entretenimiento. En particular, describiremos la importancia de una arquitectura basada en datos, justificaremos la necesidad de la utilización de código de terceros y detallaremos cómo gestionar la ejecución de todas las tareas que debe realizar la aplicación. El capítulo pasa después a explicar algunas formas de manejar las entidades que forman parte del entorno virtual, y los lenguajes de scripts como un método de sacar parte de su control a ficheros de datos.

2.1. Introducción

La construcción de programas educativos basados en juegos no es una tarea fácil. Como punto de partida, es necesaria la toma de decisiones de diseño iniciales que darán lugar a la arquitectura general del sistema. Si dejamos de lado las metodologías ágiles expuestas por la Agile Alliance (2001), esa arquitectura creada al principio del proceso de desarrollo, establecerá una serie de restricciones en el momento de la implementación.

Está claro que el arquitecto software basa sus decisiones, entre otras cosas, en su propia experiencia (parafraseando a Astrachan et al. (1998), "los buenos diseños se consiguen con la experiencia; la experiencia se consigue con los malos diseños"). En el caso particular que nos incumbe, es decir, en el desarrollo de programas educativos basados en juegos, es valiosa la experiencia en el desarrollo de arquitecturas para videojuegos, así como de programas educativos.

En este capítulo nos centraremos en el análisis de las arquitecturas de programas de entretenimiento, que servirán de base a la arquitectura de un sistema que junte este área con las aplicaciones educativas. De esta forma, se podrá comprobar qué técnicas comúnmente utilizadas en videojuegos pueden ser extrapolables a nuestro área. En el capítulo siguiente haremos lo mismo pero en el área de los programas educativos.

No existe mucha literatura relativa a la descripción de las arquitecturas utilizadas en estas aplicaciones, a pesar de ser un campo en el que cada año se crean miles de aplicaciones; como dato, en el 2006, el número de expositores en la feria del videojuego por excelencia, la E3 (*The Electronic Entertainment Expo*), superó los 500 (The Electronic Entertainment Expo, 2006).

La causa es que sólo desde hace unos años los desarrollos de videojuegos se hacen planificando por adelantado una arquitectura software. En 1999, Rollings y Morris comentaban que era muy habitual que los estudios de juegos saltaran desde el diseño del juego a su implementación directamente, sin pasar por un periodo previo de maduración de la arquitectura software que lo iba a sustentar. Con ese método de trabajo, el éxito o fracaso del desarrollo dependía casi por completo de la habilidad y nivel de experiencia de los desarrolladores.

Hoy en día la situación ha mejorado significativamente. Es difícil crear un juego AAA[1] en menos de dos años. Con estos largos desarrollos en los que se ven involucradas decenas o cientos de personas, la ingeniería del software resulta completamente necesaria. Afortunadamente, tanto los estudios como los propios desarrolladores ya se han dado cuenta de esto, como demuestra, por ejemplo, la existencia de listas de discusión dedicadas en exclusiva al tema (ver por ejemplo Sweng-Gamedev, creada en 2002).

Aún así, en muchos casos la arquitectura planeada queda restringida a un único título, y el estudio vuelve a diseñar una gran parte de ella para el siguiente desarrollo. Por ejemplo, el creador de la saga Doom y Quake, John Carmack, aseguró en una entrevista que gran parte del código de Quake 3 no se aprovechó para Doom 3, pese a pertenecer ambos al mismo género y tener jugabilidad similar (Sloan y Mull, 2003). Como ejemplo adicional, algún mensaje de la lista de distribución anterior (Sweng-Gamedev, 2002) indica que en la misma compañía se ha reescrito incluso el sistema de menús de un juego a otro, a pesar de ser un elemento independiente por completo de la temática del juego en desarrollo.

Otro problema que va de la mano de la falta de arquitectura es la frecuencia con que se dan fallos en la *planificación*, que hacen que los proyectos se retrasen numerosas veces, llegando incluso a la cancelación debido al excesivo tiempo de desarrollo. Se puede decir que, si bien los desarrolladores

[1] Se dice que un juego es AAA cuando tienen un alto presupuesto, su lanzamiento es considerado un evento mediático, etc. (Wardell, 1998).

experimentados consiguen terminar el producto, raras veces lo hacen según el calendario planificado (Fristrom, 2003), principalmente debido a que la mayor parte del código se *reescribe* de un juego a otro, y a que el éxito de la empresa se fundamenta principalmente en la experiencia individual, en vez de en un diseño formal.

Todos estos hechos pueden resumirse en que la industria de los videojuegos ha madurado mucho en los últimos años, pero los procesos de desarrollo no han avanzado a la misma velocidad. Se ha pasado desde los juegos "creados en garajes" y con poco mercado, a desarrollos de varios millones de dólares y gran repercusión mediática y popularidad. Sin embargo, los controles de la producción, gestión de los proyectos y arquitecturas software, a pesar de haber avanzado, no lo han hecho de manera acorde, por lo que no suelen estar a la altura del proyecto de la misma manera que lo están en otras áreas de la industria del software (Larsen, 2002).

A lo largo de este capítulo identificaremos los componentes básicos que aparecen en la gran mayoría de los juegos, para definir un vocabulario que utilizaremos después cuando detallemos la arquitectura propuesta para el desarrollo de aplicaciones educativas basadas en videojuegos.

Por lo tanto, el cometido de este capítulo no es proponer una arquitectura genérica que pueda ser reutilizada en el desarrollo de cualquier tipo de juegos (ese es el objetivo por ejemplo de Plummer, 2004), sino hacer un análisis de distintas arquitecturas, para poder diseñar una válida en nuestro ámbito de actuación.

2.2. Definición de arquitectura

Durante décadas, los ingenieros del software han creado sus propios diseños empezando de cero, o reutilizando diseños de otros. Hoy en día, esa situación ha cambiado, surgiendo una nueva disciplina conocida como *arquitecto software* que utiliza un lenguaje común a todos los ingenieros del software para detallar diseños de alto y bajo nivel (Braude, 2001).

La definición de lo que se entiende por arquitectura software no está, sin embargo, consensuada. Algunos autores, como Braude (2001) indican simplemente que el término "arquitectura" es equivalente a "diseño al más alto nivel" (Braude, 2001, pag. 150). También Eoin Woods habla de *decisiones de diseño* al decir que "una arquitectura software es el conjunto de decisiones de diseño que, si son incorrectas, pueden causar que el proyecto se cancele" (Rozanski y Woods, 2005).

Shaw y Garlan (1996) sin embargo, no hablan de "arquitectura" como tal, sino de "diseño a nivel de arquitectura", entendiendo éste como el relacionado con cuestiones estructurales, como la organización del sistema, protocolos de comunicación, sincronización y acceso a datos, asignación de funcionalidad a cada elemento, composición de cada uno de estos elementos, distribución

física, escalado y rendimiento, etc.

Cuando se habla de arquitectura como un diseño de alto nivel, tiene sentido hablar de patrones arquitectónicos (llamados "estilos arquitectónicos" por Shaw y Garlan (1996)), arquitecturas por capas, tuberías y filtros, pizarras, o modelo-vista-controlador (Buschmann et al., 1996).

Una definición más apropiada es la expuesta por Bass et al. (2003), en la que definen la arquitectura de un programa o sistema de ordenador como la estructura o estructuras del sistema, que comprenden los elementos software, las propiedades de esos elementos visibles desde el exterior y las relaciones entre ellos.

Esta forma de entender una arquitectura software es compartida por Rollings y Morris (2004), y también es la que aceptaremos de aquí en adelante. Resumiendo, *una arquitectura software es la estructura de un programa, y abarca también el flujo de datos, definiendo las interacciones entre todos los elementos.*

2.3. Arquitecturas software en juegos

Como ya queda dicho, la complejidad de los juegos ha llegado a unos límites inimaginables hace sólo unos años. Hoy por hoy, para conseguir un juego de alta calidad, se requiere un presupuesto capaz de soportar un gran número de personas trabajando durante al menos año y medio. Para corroborarlo, basten los siguientes datos:

- En Noviembre de 1999 salía Unreal Tournament, después de 18 meses de trabajo de 16 personas, un presupuesto de dos millones de dólares y 350.000 líneas de código entre C++ y UnrealScript (Reinhart, 2000).
- El juego "Vampire: the Masquerade" requirió 12 desarrolladores durante 24 meses, casi dos millones de dólares y 366.000 líneas de código (Huebner, 2000).
- Peter Molyneux puso a la venta en Marzo de 2001 su juego "Black & White", después de invertir casi seis millones de dólares en 25 desarrolladores durante tres años, que escribieron dos millones de líneas de código (Molyneux, 2001).
- En Junio de 2002 salío a la luz Neverwinter Nights, después de 5 años de desarrollo con un equipo que en algunos momentos llegó a ser de hasta 75 personas (Grieg et al., 2002).
- En 2003, sólo portar el Splinter Cell de la plataforma XBox a Play Station 2 requirió 76 personas durante cinco meses a tiempo completo (Hao, 2003).

- El desarrollo de juegos "de nueva generación" para las nuevas consolas maneja cifras aún más escalofriantes. Para el Project Gotham Racing 3 de XBox 360, tuvieron que invertir 80.000 libras sólo para comprar servidores en los que almacenar el arte del juego, y las veinte mil fotografías en las que se basaron para modelar cada una de las ciudades que aparecían en él (Gillen, 2005).
- El desarrollo de Call of Duty 2 para XBox 360 requirió 2 años de trabajo de 75 personas, y tuvo un presupuesto de más de 14 millones de dólares (Duffy, 2005).

Al ver estas cifras, queda claro que el desarrollo de juegos ha cambiado desde aquellas primeras aplicaciones desarrolladas para consolas, recreativas y ordenadores de 8 bits.

Desde siempre, este tipo de aplicaciones han intentado aprovechar hasta el último recurso del *hardware* para hacer más llamativo el aspecto final del juego. Estos programas son aplicaciones *de tiempo real*, donde una respuesta dada tarde puede romper toda la sensación de inmersión del jugador. Es necesario que el número de fotogramas mostrados por segundo sea elevado, para que el entorno presentado se considere interactivo y animado. Es por ésto que los programadores deben conocer a la perfección el funcionamiento interno de todos los componentes para no desperdiciar recursos debido a una mala programación. Cuando los recursos disponibles crecen, no decrece el esfuerzo necesario para crear los juegos. Al contrario, lo que ocurre es que aumentan sus pretensiones, lo que hace que la complejidad de programación, generación de modelos y recursos en general, también aumente. Esto conlleva un crecimiento notable tanto en el tiempo de desarrollo de un único título como en el aumento del número de personas del equipo.

Desde el punto de vista histórico, durante la década de los 80 y principios de los 90, era habitual que los juegos se siguieran desarrollando en ensamblador, a pesar de los grandes avances en los compiladores. La idea generalizada era que el compilador no podía competir con un programador de ensamblador experto, por lo que los desarrollos seguían utilizándolo a pesar de aumentar significativamente la duración del proyecto. Durante la década de los 90 la situación, por fin, fue cambiando, y los estudios comenzaron a utilizar activamente lenguajes de alto nivel, especialmente C debido a sus capacidades de bajo nivel. Entre los compiladores estrella de aquella época hay que destacar, en el mundo del PC, Watcom C. Este compilador fue ampliamente utilizado a mediados de los 90, para la programación de juegos para MS-DOS (como Doom). Su éxito radicó en una muy buena calidad del código generado, y en su soporte para la programación en modo protegido del Intel 386. Desgraciadamente, Sybase, la empresa que lo creó, abandonó el proyecto en 1999, ya que su principal mercado era el del mundo del PC (en Windows), y no podían competir con Microsoft.

Se puede decir que en aquella época se vivió un gran salto en el proceso de desarrollo, ya que se pasó de programar la práctica totalidad del juego en ensamblador, a utilizar lenguajes de alto nivel. Pensando fríamente, constituyó un gran paso adelante, ya que los programadores tuvieron que rendirse a la evidencia de que era mejor aceptar que el código ejecutado por la máquina durante el juego fuera generado por un compilador, que alargar un proceso de desarrollo debido al uso del tedioso y propenso a errores aunque óptimo ensamblador.

Lo que se tardó mucho más en aceptar fue el uso de código desarrollado por otros (*código externo*) en el propio juego[2]. Cada estudio, una y otra vez, programa parte de código que podría haber *comprado* a desarrolladores externos. Es lo que en el mundo anglosajón se conoce como el *not invented here syndrome*, o reticencia a utilizar algo no creado en el propio estudio. Esta reticencia se debía principalmente al hecho de pensar que el estudio podía desarrollar ese módulo o funcionalidad de forma más óptima que el componente comprado.

Personalmente, estimo que una de las cosas que ayudó a dar el salto, y a que se empezara a aceptar el incluir desarrollos externos en el propio juego fue la explosión de tarjetas gráficas aceleradoras. Pronto se hizo evidente que los desarrolladores no podían optar por la programación a bajo nivel del motor gráfico, accediendo directamente a las capacidades de la tarjeta. Esto era debido a que el número de tarjetas disponibles crecía rápidamente. En este contexto surgió Windows 95, que además imponía limitaciones de acceso al *hardware*. Afortunadamente, también proporcionaba las librerías DirectX y OpenGL. Éstas colocaban (y aún colocan) una barrera de abstracción entre el *hardware* y la aplicación, de tal forma que el programador no tenía que lidiar con las diferencias entre las distintas tarjetas. Las amplias ventajas de esta aproximación hicieron que los estudios incorporaran estas librerías (externas) en sus juegos rápidamente.

Sea cual sea la causa, poco a poco los estudios de desarrollo han ido aceptando la inclusión de código externo en forma de librerías o módulos completos comprados a terceros. Son lo que en inglés se conoce como COTS (*Component off the shelf*).

2.3.1. Razones para el uso de COTS

En realidad, el uso de COTS es consecuencia de la imposibilidad de reutilización del código en un mercado que evoluciona tan rápidamente como el de los videojuegos.

El problema de la reusabilidad es que ésta sólo es posible si el software

[2]No es de extrañar, si se tiene en cuenta todo lo dicho anteriormente: si los estudios no se fiaban del compilador, y eran reacios a reutilizar su propio código entre proyectos, mucho más difícil les resultaba utilizar código de otros.

desarrollado se construyó pensando en volver a ser usado (Tracz, 1995a); es decir, no vale con coger el código creado para el título anterior, y utilizarlo directamente. Es necesario que ese código que se quiere volver a usar se creara con la reutilización en mente.

Durante la década de los 90, cuando nosotros situamos la migración de los estudios desde el ensamblador hacia los lenguajes de alto nivel, el área de la reutilización estaba en auge. Se dedicó, y aún se dedica, mucho esfuerzo para generar aplicaciones que automaticen o al menos ayuden en la reutilización del código. Sin embargo, Tracz (1995b) nombra un estudio que revelaba que el coste de desarrollo del código se ve incrementado entre un 30 y un 50 por ciento si se implementa con miras a la reutilización. Además, el diseño de las clases nuevas hay que hacerlo con cautela, para no caer en la *sobreingeniería* (Kerievsky, 2002), lo que provocaría una pérdida aún más excesiva de recursos.

Por poner un ejemplo[3] de la razón de este incremento, se puede pensar en un juego que requiera un dado con los números del 1 al 6 en sus caras. La implementación de una "tirada", requiere una simple línea en C/C++, utilizando la función `rand()`, el módulo y el incremento. Sin embargo, implementar un código reutilizable de un dado implica mucho más tiempo (y por tanto, dinero). En primer lugar, es posible que el dado deseado en un futuro no tenga seis sino más caras, por lo que el "módulo" o clase `CDado` deberá tener un parámetro para indicar el número de caras. Además, con un número de caras variable, lo que aparece en cada una de ellas debería ser configurable. El código de lo que antes era una mera línea, ahora se ha convertido en un fichero de definición y otro de implementación de una clase que habrá que programar y probar, para garantizar su correcto funcionamiento. Cuando este proceso esté completo (o mientras se completa), también habrá que escribir la documentación pertinente para facilitar la reutilización.

Pero no todo son aspectos negativos; el notable incremento del coste en la escritura del código reutilizable se compensa con el abaratamiento de los desarrollos posteriores, que simplemente tienen que hacer de *usuarios* de ese software. De esta forma el esfuerzo dedicado en la construcción de clases o módulos reutilizables se convierte en una inversión del estudio a medio o largo plazo.

Sin embargo, el problema en la industria del videojuego es, precisamente, la dificultad de pensar a largo plazo, debido a que la ventana de mercado es muy corta. La causa es el rápido avance de las capacidades del hardware[4], y el hecho de que los juegos que se lanzan al mercado deben aprovechar los últimos avances tecnológicos para resultar atrayentes al consumidor.

[3] Atribuido en Adolph (1999) a Luke Hohnmann.

[4] Obviamente, el ejemplo del dado en este caso no nos sirve. Para fundamentar esto, estamos pensando en la reutilización de componentes más grandes, como el motor gráfico o el de física.

Por lo tanto, existen dos barreras que frenan la generación, por parte de los estudios de juegos, de código altamente reutilizable:

- Motivo económico: la creación de código y componentes pensando en su reutilización es más costosa.
- El propio mercado: por dos razones; (i) la creación de estos componentes generalmente retrasa la salida del primer título que los utiliza, algo que puede ser de vital importancia, al existir el riesgo de quedarse desfasado tecnológicamente; (ii) la rápida evolución del mercado hace que muchos componentes pensados para su reutilización queden obsoletos antes de poder volverlos a usar.

Únicamente los estudios grandes que disponen de varios proyectos en paralelo pueden plantearse la posibilidad de desarrollar internamente partes del juego que reutilizarán. En ese caso, al salir al mercado varios títulos en un corto espacio de tiempo, los componentes que se reutilizan no han quedado desfasados de un juego a otro.

El resto de estudios, en general, no pueden hacer frente a la creación de componentes reutilizables. Eso, unido a la necesidad de reducir los tiempos de desarrollo, hace que las prácticas software de dichas empresas están migrando desde los desarrollos en los que todo el código era construido por el propio estudio, a la *compra* de componentes software implementados por terceros.

De esta forma, surgieron empresas como RenderWare[5], que desarrolla un motor gráfico independiente de la plataforma utilizado por más de 500 juegos entre los que destacan Grand Theft Auto: San Andreas o Call of Duty: Finest Hour, y como Havok[6] que tiene una familia de utilidades, como un motor de animación o de física, y que se ha utilizado en juegos como Half-Life 2 o F.E.A.R.. Conviene también destacar el caso especial de idSoftware y Epic Games, dos estudios de juegos que, si bien la calidad de los mismos en cuanto a guión puede ser discutible, la calidad gráfica que consiguen es indiscutible. Sus juegos pueden entenderse como "propaganda" de un producto "derivado" que también comercializan: la tecnología gráfica. Ambos estudios *licencian* sus motores gráficos a terceros, que los incorporan a modo de COTS en sus desarrollos. Por nombrar un ejemplo, Epic Games ha licenciado su motor a los estudio de Microsoft y Disney.

2.4. Arquitecturas dirigidas por datos

Inicialmente, el tamaño de los juegos era muy pequeño, el contenido de la parte gráfica era mínima, y lo único importante era sacar el máximo

[5] `http://www.renderware.com`
[6] `http://www.havok.com`

a) Captura del Laberinto del Sultán (Amsoft, 1984).

```
660 MOVE 320,300: DRAW 320,351,1
670 MOVER 9,0: DRAWR 43,-12,0:
    DRAWR 0,-4: DRAWR -43,-4:
    DRAWR 0,20: MOVER 0,-2:
    DRAWR -4,0: MOVER 0,-16:
    DRAWR 4,0
680 MOVER 4,0: DRAWR 0,16,1:
    MOVER 4,-2: DRAWR 0,-14: ...
```

b) Fragmento de código original del juego

Figura 2.1: Ejemplo de juego con datos acoplados en el código

rendimiento a unos escasos recursos hardware para conseguir una aplicación que resultara mínimamente divertida.

Con estos requisitos y máquinas (estamos hablando de la primera mitad de la década de los 80), los juegos se hacían principalmente en ensamblador, y los datos necesarios (como las posiciones de los muros del Comecocos, o la forma de pintar los fantasmas) estaban *cableados* en el código. Por poner un ejemplo real (aunque no sea en lenguaje ensamblador), la figura 2.1 muestra en su parte izquierda una captura del juego "El Laberinto del Sultán", desarrollado por Amsoft en 1984. El dibujo está hecho completamente desde el código, en este caso BASIC. Un fragmento de este código puede verse en la parte derecha de la figura, en la que todas las instrucciones de dibujado (`DRAW`, `DRAWR`, `MOVE`, `MOVER`) reciben como parámetro las coordenadas[7].

Otra forma de acoplamiento entre código y datos se daba con la existencia de los llamados *segmentos de datos* (`.data`) en el propio ejecutable que almacenan los datos referenciados desde el código.

Sea como fuere, este fuerte acoplamiento entre el comportamiento del juego (lógica) y los datos necesarios para su ejecución, fue separada mediante el uso de ficheros. Ya durante la década de los ochenta aparecen juegos cuyo núcleo principal (ejecutable), una vez cargado en memoria, leía ficheros de datos de los dispositivos de almacenamiento (disco duro, disquettes o incluso cintas), de los que obtenía la información para dibujar al personaje, el mapa del nivel o las posiciones de los enemigos. Esta separación permitía que el trabajo de desarrollo pudiera dividirse entre la tarea de los programadores, que eran los responsables últimos del ejecutable, y los artistas, que creaban los ficheros con la parte gráfica.

Hoy por hoy esta división es tan habitual que ya nadie piensa que en

[7]Estas instrucciones operan con el "cursor gráfico", moviéndolo, o dibujando líneas desde su posición hasta la indicada en el parámetro.

otros tiempos no fue así, y se ha pasado a una separación aún mayor: parte del propio *ejecutable*, o mejor dicho, parte del *código que controla el juego* se ha sacado fuera del fichero ejecutable propiamente dicho. Esta separación es posible gracias al uso de librerías de carga dinámica proporcionadas por el sistema operativo[8] o mejor aún, el uso de lenguajes de scripts, que veremos en la sección 2.7, como LUA (Ierusalimschy et al., 2006), Python (van Rossum, 2006) o UnrealScript (Sweeney et al., 2004).

Este modelo ha hecho que los programadores, al contrario de lo que sucedía a principios de la década de los 90, han dejado de ser el centro del desarrollo de juegos (Llopis, 2005a). Hoy en día, el rol del programador es muy distinto: el *juego* es creado por los artistas y diseñadores, que son los que construyen el *contenido* (en inglés, "*game content*"). De esta forma, su tarea se ha desplazado desde ser el centro del desarrollo a estar al servicio del resto del equipo, proporcionándole los medios necesarios para crear un buen juego.

Debido a la separación clara entre código y datos, han surgido distintas áreas de programación claramente diferenciadas en los estudios:

- Motor del juego: código que va en el juego final, pero que no es específico del juego en cuestión. Es la base de código sobre la que se sustenta el juego en sí; se apoya en la tecnología subyacente, y simplifica el desarrollo del resto de la aplicación. Además, en muchas ocasiones, permite la ejecución del juego en distintas plataformas, como PC y consolas. Su funcionalidad típicamente pasa por el motor gráfico (para presentar las escenas 2D y 3D), un motor de física o de colisiones, gestión del sonido, animación, lenguajes de scripts y red. Dada su complejidad, éste es el ámbito de actuación principal de los COTS.

- Código del juego ("*game code*"): la parte de código específica del juego particular; por ejemplo, cómo reaccionan los personajes controlados por el ordenador ante ciertas situaciones, cómo se comporta la cámara, o cómo se almacenan los puntos del jugador. Esta parte del código es muy importante, y en muchos casos se realiza, al menos parte de ella, en lenguajes de script.

- Herramientas: cuanto más orientada a datos es la arquitectura de un juego, más ficheros de datos hay que generar, y por lo tanto, más herramientas para ayudar en esa generación se necesitan. En cuanto a herramientas, pueden ser "simples" plug-in para otras aplicaciones (como Maya o 3DStudio Max), o programas complejos para edición de mapas (por ejemplo, Worldcraft/Hammer, UnrealEd y Sandbox). El lenguaje utilizado para crear estas herramientas no tiene por qué ser C/C++.

[8] DLLs en Windows o `.so` en GNU Linux/Unix

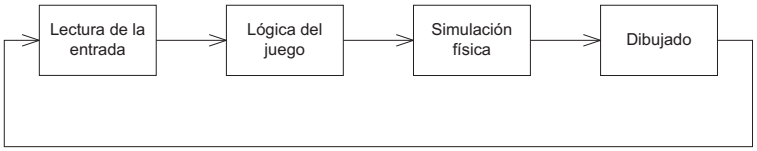


Figura 2.2: Esquema de bucle principal básico

2.5. Bucle principal

La ejecución de un juego está dirigida por lo que se conoce como el *bucle principal* (en inglés *game loop*), que es el responsable de la ejecución de cada una de las tareas requeridas en cada fotograma, y que van encaminadas a crear la ilusión de un entorno *vivo* y animado. A pesar de que la programación del bucle principal lleva poco tiempo, su importancia es vital, ya que es el eje conductor que va estableciendo qué módulos se ejecutan, cuándo y cada cuánto tiempo. El modelo de bucle principal utilizado define, por ejemplo, si la lógica avanza en intervalos constantes o no, si la entrada del usuario es analizada casi instantáneamente o su procesamiento es retrasado varios fotogramas, etc.

La lista de tareas que debe realizar el bucle principal (ver figura 2.2) son:

- Gestión de la entrada del usuario: dado que un juego es una aplicación inherentemente interactiva, es importante que la lectura de la entrada se realice de forma correcta. Puede involucrar desde simplemente comprobar si se ha pulsado alguna tecla, a la gestión de complicados dispositivos de entrada.

  En vez de distribuir la lectura de la entrada por las tareas que dependen de ella (por ejemplo, mirar el estado de las teclas cuando se va a actualizar la posición del usuario y comprobar los eventos del ratón cuando se va a actualizar la cámara), se suelen analizar todos los dispositivos de entrada a la vez, almacenando el resultado de esas lecturas. Esos resultados son los que luego consultan las distintas tareas en el momento de su ejecución. De esta forma, se garantiza que la aplicación reaccionará de manera consistente en todo el fotograma.

  Así, la lectura de la entrada pasa a ser una tarea por sí misma, que suele situarse al principio del bucle de la aplicación. En general, se ejecuta una vez en cada fotograma (o "vuelta del bucle"), aunque en juegos en los que la interacción no es tan importante (como los de estrategia) puede hacerse cada varios fotogramas.

- Gestión de la red: en juegos multijugador, la red puede verse como otra forma de entrada, por lo que en algún momento del bucle de juego se

deben recopilar todos los mensajes de ésta y procesarlos.

- Simulación o actualización de la lógica del juego: es donde el juego realmente avanza. La simulación se encarga de decidir los comportamientos de los personajes (IA), ejecutar las acciones de los mismos, actualizar el estado de los objetos, lanzar eventos que provocarán otras futuras acciones, etc.

  También se encarga de que los personajes controlados por el jugador reaccionen ante los eventos recogidos en las fases anteriores.

- Simulación física y gestión de colisiones: se encarga de mover los objetos de acuerdo con la simulación física subyacente (velocidad y aceleración actual), y puede encargarse de actualizar los sistemas de partículas o el avance de las animaciones de los personajes. La simulación física en juegos sencillos puede llevarse a cabo dentro de la etapa anterior de simulación de la lógica del juego, aunque hoy por hoy se tiende a separar. La ejecución de la física puede provocar que los objetos simulados colisionan entre sí. En ese caso, se debe reaccionar ante esas colisiones, tanto siguiendo las propiedades físicas del mundo (por ejemplo, haciendo que una caja rebote en una pared) como las lógicas (haciendo que el jugador pierda energía si una bala colisiona contra él, etc.).

- Dibujado del mundo: todas las etapas anteriores van encaminadas a calcular el nuevo estado del entorno virtual: las nuevas posiciones de los objetos, sus estados de animación, etc. En esta última fase del bucle principal, este estado recién calculado es utilizado para dibujar la nueva escena.

Existen algunas otras tareas de menor importancia que no han sido situadas en ninguna de las fases del bucle, y que, no obstante, no deben olvidarse, como son la actualización del sistema de sonido para que haga todas las mezclas necesarias, el procesado de los eventos enviados por el sistema operativo a la aplicación/ventana (para reaccionar, por ejemplo, ante cambios de contexto, activación del protector de pantalla, o entrada en modo ahorro de energía), o el envío de paquetes de red construidos durante la simulación.

En general, se puede decir que cada vuelta del bucle al final tiene como resultado la creación de un nuevo fotograma[9]. Generalmente, se utiliza como parámetro para medir el rendimiento el número de vueltas por segundo que se ejecuta el bucle, o lo que es lo mismo, el número de fotogramas por segundo (FPS) que se dibujan en pantalla. Como límite inferior para poder decir que la aplicación es interactiva, suelen aceptarse los 16 fotogramas por segundo

[9]En sistemas multihebra, el concepto de "bucle principal" se difumina, ya que cada hebra puede ejecutar un bucle distinto, y sólo uno de ellos termina con el dibujado de un fotograma.

(Valente et al., 2005), aunque el óptimo suele estar entre 50 y 60 (que suele coincidir, o al menos acercarse, a la velocidad de refresco del monitor).

Existen dos alternativas básicas en cuanto al control de los fotogramas por segundo:

Duración del fotograma variable El tiempo que transcurre entre la presentación de un fotograma y el siguiente varía dependiendo de la situación de juego. En momentos en los que se requiere gran cantidad de cálculos (por picos en la inteligencia artificial o física, o gran número de elementos a dibujar), los fotogramas por segundo pueden bajar, mientras que en otros momentos, pueden subir, aumentando la fluidez de las animaciones, etc.

Duración del fotograma fija El número de fotogramas por segundo es siempre el mismo. Esto es más fácil de conseguir cuando el *hardware* al que va destinado el juego está predeterminado y es fijo. El ejemplo más claro son las consolas. En ellas, los fotogramas se sincronizan con la actualización del sistema de video (25 fps en PAL y 30 en NTSC). En este caso, las tareas a ejecutar en el bucle principal deben asegurarse de que no bloquean la CPU más tiempo del disponible.

Por otro lado, la lógica del juego es la que hace avanzar el estado del entorno, simulando el paso del tiempo y el comportamiento de todas las entidades que lo pueblan. Para su actualización, existen también dos formas básicas de simular el paso del tiempo, que tienen sentido especialmente cuando se utiliza una duración variable de cada fotograma:

Paso de tiempo variable Cada vez que se va a ejecutar la tarea de la lógica/simulación, se comprueba, utilizando el reloj del sistema, cuánto tiempo ha pasado desde la actualización anterior. El código recibe mediante un parámetro ese tiempo, y actúa en consecuencia.

Este método de funcionamiento es el más intuitivo y comúnmente aceptado. No obstante, presenta algunas desventajas. Entre ellas, cabe destacar lo complicado que resulta corregir los errores detectados, al ser difícil de reproducir una ejecución anterior. También es cuestionable que en ciertos juegos la lógica se esté ejecutando una vez cada fotograma. Por ejemplo, en un juego de estrategia en principio no es necesario plantearse las acciones a ejecutar 60 veces por segundo.

Paso de tiempo fijo Independientemente del tiempo *real* (o "*físico*") que haya pasado desde la última actualización de la lógica, la simulación se realiza considerando que el lapso de tiempo es siempre el mismo. Si el ordenador es lo suficientemente rápido, puede darse el caso de que el bucle principal se ejecute en menos tiempo que el lapso de tiempo

```
time0 = GetTime();
do {
  time1 = GetTime()
  int numLoops = 0;

  // Tareas independientes de la simulación
  // (gestión de entrada, etc.)
  IndependentTickRun();

  // Simulación
  while ((time1 - time0) > TICK_TIME &&
         numLoops < MAX_LOOPS) {
    GameTickRun(); // No es necesario parámetro de tiempo.
    time0 += TICK_TIME;
    numLoops++;
  }

  // Dibujado con interpolación
  float porcentajeDeTick;
  porcentajeDeTick = min(1.0f,
                         float(time1 - time0)/TICK_TIME);
  GameDrawWithInterpolation(porcentajeDeTick);
} while (!bGameDone);
```

Figura 2.3: Bucle principal con fotograma variable y tiempo fijo

de simulación; en ese caso el bucle no actualiza la lógica, sino que se limita a dibujar. En ordenadores lentos puede ocurrir la situación inversa, exigiendo al bucle actualizar la lógica varias veces por vuelta.

Obviamente, desacoplar la simulación y el dibujado implica algunas acciones más. Si ejecutamos la simulación 15 veces por segundo, pero mostramos 30 fotogramas, podemos estar duplicando el *frame* anterior. Para evitarlo, antes de dibujar, habrá que hacer una interpolación de la posición y rotación de los objetos, así como la actualización de su animación; para ello, habrá que añadir una fase posterior a la simulación y comprobación de colisiones que sea la *interpolación del último estado* generado por la lógica. Para que no surjan problemas, se debe intentar evitar las colisiones que pueden darse en la interpolación de ese estado. Otra alternativa es hacer que el estado de la lógica vaya *por delante* del estado que se está dibujando. De esta forma, al dibujar, se *extrapola* hacia atrás la lógica, de tal forma que el problema de las posibles colisiones se elimina.

La figura 2.3 muestra el código C++ simplificado de un bucle principal que utiliza este mecanismo de paso de tiempo fijo Arévalo (2001). Al principio, ejecuta operaciones que no dependen del tiempo, como

lectura de la entrada y otras operaciones de mantenimiento. Después ejecuta la simulación. Como se ve, ésta sólo se ejecuta si el tiempo transcurrido desde la última simulación (`time1 - time0`) supera el *tiempo de simulación* (`TIME_TICK`). Por último, se realiza el dibujado del entorno, realizando la interpolación de los elementos desde el momento de la última simulación.

2.5.1. Bucles multihebra

Hoy en día, el *hardware* ha avanzado lo suficiente como para permitir la ejecución de hebras distintas con paralelismo *real*. Por ejemplo, en los PCs empieza a ser habitual ordenadores capaces de ejecutar dos o cuatro hebras; la consola de Microsoft XBox 360 tiene un procesador con tres núcleos, cada uno con hyperthreading, y la Play Station 3 de Sony tiene un procesador central y siete procesadores satélite.

Esto hace que los arquitectos software deban plantearse la paralelización de todos los módulos del juego, para aprovechar todas estas capacidades. En el caso del PC y de la consola XBox, esta paralelización es relativamente sencilla (son sistemas multiprocesador *simétricos*, SMP). Sin embargo, en la Play Station 3, las divisiones de los distintos módulos para su paralelización es más complicada. Una introducción al problema puede encontrarse en Gabb y Lake (2005), mientras que Mönkkönen (2006) propone varias soluciones. También es destacable la aportación de Rhalibi et al., en la que sugieren por primera vez el uso de grafos de dependencias de tareas para el desarrollo de juegos con ejecución paralela (Rhalibi et al., 2006). Por último, se puede consultar Turner (2007) como ejemplo real de un planificador de tareas diseñado para optimizar el uso de 6 hebras *hardware* en el juego Saints Row (Volition Inc., 2006).

2.6. Entidades del juego

Un juego, o mejor, un entorno virtual gira en torno a la interacción entre el usuario y el mundo. Debido a esto, podemos decir que existen dos categorías de objetos dentro del entorno: los objetos dinámicos y los estáticos.

Los objetos estáticos son inmutables, y el usuario no puede hacer nada con ellos, excepto verlos y chocarse contra ellos. Ejemplos son muros, rocas, etc.

Los objetos dinámicos son aquellos que hacen del entorno un mundo *vivo*, que cambian sus propiedades durante la ejecución de la aplicación, ya sea debido a la interacción del usuario o de otros objetos. En el contexto de los juegos, estos tipos de objetos suelen conocerse como *entidades*, aunque también pueden denominarse "objetos de juego" (en inglés, *game objects*) (Plummer, 2004).

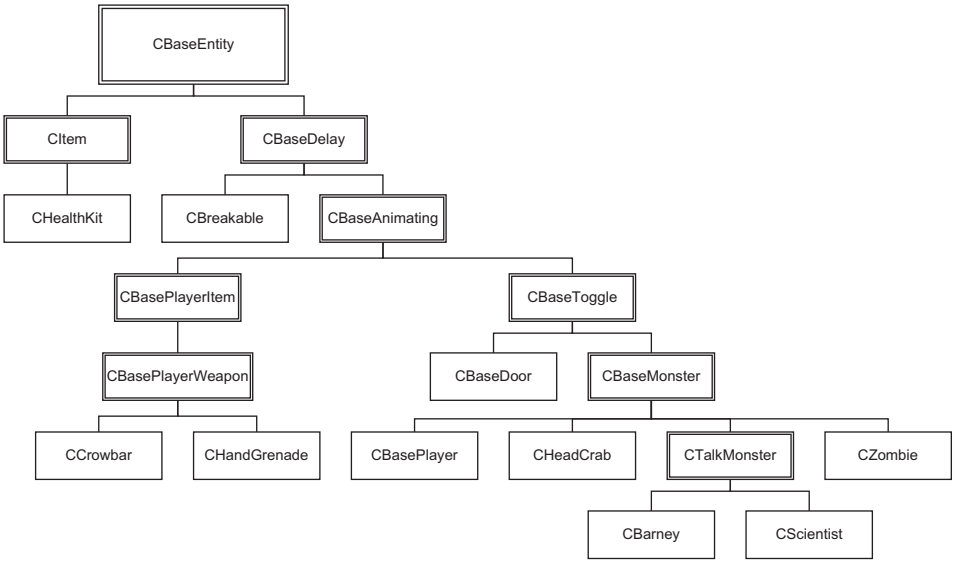


Figura 2.4: Jerarquía parcial de los tipos de entidad de Half-Life 1

No existe un consenso en la definición de lo que es una entidad, pero aquí nos quedaremos con la dada por Llopis (2005b), que la define como "una pieza autocontenida de contenido interactivo".

Dada la naturaleza de las entidades, es natural aplicar el paradigma de orientación a objetos para codificar su comportamiento. De esta forma, es habitual disponer de una jerarquía de clases para representar los distintos tipos de entidades. Por ejemplo, podemos tener la clase `CEntity`, de la que hereda una clase `CMoveable`, que represente a cualquier entidad que puede moverse por el entorno. Por debajo de ésta, tenemos `CVehicle`, `CHuman`, `CGun`, etc. De la clase que representa cualquier avatar, heredarían los distintos enemigos y el propio jugador.

Como ejemplo real, la figura 2.4 presenta parcialmente la jerarquía de clases utilizada en el Half-Life 1 (Valve Software, 1998).

Todas las entidades heredan de `CBaseEntity`, de la que parten, entre otras clases, la clase que representa los objetos que pueden cogerse (*items*) que aparecen en el juego (en la práctica, únicamente se implementa la clase `CHealthKit`). También aparecen las clases `CBaseDelay` y `CBaseAnimating` que extienden la funcionalidad básica de la clase base, aunque aún no especifican por completo su comportamiento. De la última parten ya todos los items que el jugador puede llevar, en particular, todas las armas. También de `CBaseAnimating` hereda `CBaseToggle`, que permite a todas sus subclases "cambiar de estado". Hijas de esta son las puertas y todos los enemigos del juego.

Los juegos se convierten en una "base de datos de entidades con un interfaz bonito", en la que el sistema de objetos almacena todas las entidades y las gestiona. En el siguiente apartado describimos cómo organizar las entidades teniendo presente su actualización. Posteriormente trataremos sobre la construcción de las entidades, y terminaremos el apartado con la explicación de la arquitectura basada en componentes, la forma de estructurar las entidades u objetos del juego que se está imponiendo en la construcción de videojuegos, y que utilizamos en nuestra arquitectura para el desarrollo de aplicaciones educativas basadas en videojuegos que describimos en el capítulo 5.

2.6.1. Organización y actualización de entidades

Un videojuego tiene en todo momento una lista de todas las entidades disponibles en el entorno virtual. Dado el carácter variable y volátil de las entidades (se disparan balas, con tiempo de vida muy corto, los enemigos mueren y nacen otros, etc.), resulta mucho más adecuado almacenarlas en una lista enlazada que en un vector estático. De esta forma la ejecución de sus operaciones es mucho más eficiente.

La implementación intuitiva de la fase de actualización de la lógica o simulación del bucle principal que veíamos en el apartado 2.5 consiste en recorrer esa lista de entidades y llamar a un método de actualización, en el que se cede a la entidad el control de la CPU para que altere sus propiedades.

En la práctica, este recorrido lineal es muy ineficiente, ya que gran cantidad del tiempo las entidades no tienen que realizar ningún tipo de actividad durante su actualización. Mientras tanto, el motor del juego insistirá en recorrer todos los elementos de una lista enlazada (con los fallos en la caché de datos que eso provocará) para llamar al código del método `update` de la clase correspondiente (de nuevo provocando fallos de caché, esta vez en la de código), para encontrarse un simple `return true;`.

Para solventar esta ineficiencia, los distintos juegos utilizan alternativas diferentes. De esta forma, los juegos de estrategia suelen utilizar un mapa basado en una rejilla, y almacenan por cada uno de sus componentes, las entidades que hay en él. Pritchar (2001) describe cómo utilizar una técnica similar a esta para un cálculo rápido de las entidades que entran dentro del campo de visión del jugador.

El género de los FPS (*first-person shooter*, juegos en primera persona en donde el jugador va con un arma disparando a los enemigos), lo que se hace es dividir todo el espacio en regiones volumétricas, cada una de ellas con la lista de entidades que contiene. Estas regiones se pueden estructurar de dos formas: o bien utilizando portales (Luebke y Georges, 1995), que vienen a ser "habitaciones" y "puertas" dentro del entorno virtual, o árboles BSP (*Binary Space Partitioning*, Particionado binario del espacio), que es una técnica más general que la anterior pero más complicada de implementar (Foley et al.,

1990; Abrash, 1997). También se pueden utilizar quadtrees y octrees (Finkel y Bentley, 1974), que son técnicas independientes de la geometría del entorno.

En aplicaciones con un gran número de entidades, la gestión de cuáles están activas y cuales no es de vital importancia, para ahorrar tiempo a la CPU que evita invertir recursos en la actualización de aquellas que no lo necesitan. Con el incremento de memoria, hoy por hoy es habitual que la aplicación maneje varias listas de entidades, para optimizar distintos recorridos. Así por ejemplo, puede tener una estructura en portales o un árbol BSP para buscar rápidamente las entidades que aparecen cerca del usuario o para descartar rápidamente las entidades que no hay que dibujar, pero utilizar una cola de prioridad en la que las entidades más importantes (por cercanía al jugador o por relevancia en el juego) aparecen antes, y que es recorrida en la etapa de actualización de la lógica. Para ello, se puede utilizar un sistema de tareas con distinta prioridad (McLean, 2002), o *micro hebras* manejadas por el propio juego, que cede una cantidad de tiempo limitada para la actualización de cada entidad (Dawson, 2001).

2.6.2. Construcción de las entidades del juego

Si tenemos una clase por cada tipo de entidad, la creación de un objeto de estas entidades es, en principio, tan simple como utilizar el operador `new` de C++ con esa clase.

Sin embargo, la información sobre qué entidades hay que crear *no* se conoce en tiempo de compilación del juego, sino que se lee de ficheros externos (mapas). Por esto, se necesita un mecanismo capaz de crear objetos que representen a entidades a partir de los datos almacenados en disco.

Normalmente, el fichero almacenará el nombre (o clase) de la entidad y las propiedades de construcción, como la posición o el modelo 3D. A partir de ella, habrá que construir el objeto particular.

La alternativa obvia es utilizar el patrón *factoría* (Gamma et al., 1995). Si consideramos como clase base de todas las entidades la clase `CEntity`, el método de creación de la factoría recibiría un identificador del tipo de objeto a crear, y devolverá un puntero a un objeto de esa clase. La figura 2.5 muestra el código simplificado.

Sin embargo, esta aproximación tiene algunos problemas:

- El programador de la clase factoría tiene que conocer en el momento de la creación de la clase, qué objetos se van a poder crear (es decir, están programadas directamente las entidades que se podrán generar).

  Aunque esto no parece un problema a primera vista, si tenemos en cuenta la forma de estructurar el código descrita en el apartado 2.4, el cargador del mapa está situado en el *motor del juego*, y es independiente de las entidades específicas del juego en cuestión. Por lo tanto, tener

```
enum EntityType {
   PLAYER, ENEMY, POWERUP, WEAPON,
   // ... Resto de tipos de entidades
};

CEntity *CFactoriaEntidades::Create(EntityType type) {
   switch (type) {
      case PLAYER:  return new CPlayer();
      case ENEMY:   return new CEnemy();
      case POWERUP: return new CPowerup();
      case WEAPON:  return new CWeapon();
      // ...
   }
   return NULL;
}
```

Figura 2.5: Ejemplo sencillo de factoría de creación de entidades

fijado el tipo de entidades que vamos a poder crear dificulta la adición posterior de otros objetos.

Un problema menor aparece durante el desarrollo del juego, ya que añadir nuevas entidades requiere añadir una nueva entrada al tipo enumerado, y un nuevo caso en la instrucción `switch`.

- En algunas ocasiones, lo que en el mapa leído del disco es una entidad, en el código se corresponde con la mera ejecución de una serie de instrucciones que no necesariamente crean un objeto. Eso es debido a que el conjunto de entidades disponibles a la hora de editar el nivel no tiene por qué coincidir con el que se utiliza para gestionar los objetos del juego. Durante la edición del nivel existen entidades que únicamente son colocadas para *crear o inicializar* datos en los servicios del motor del juego, como el sistema de colisiones, o la IA. Por ejemplo, el diseñador del nivel puede colocar lo que para él son entidades (y que como tal se graban en el fichero del mapa) que indican los nodos del grafo para la búsqueda de caminos. Cuando en tiempo de carga de ese nivel, el juego se encuentra con tales entidades, lo único que tiene que hacer es añadir la posición que ocupa esa "entidad" como vértice del grafo, sin necesidad de crear ningún objeto "`CNodoGrafo`".

Para resolver el primero de los problemas, se utiliza una *factoría extensible*. La idea de esta factoría es que permite registrar y deregistrar los tipos de objetos que es capaz de crear. Para crearlos, delega en los verdaderos *constructores* de cada una de las clases, indicados a la hora de registrar el nuevo tipo de objetos.

```
class CEntity;

class CConstructorEntidad {
public:
    virtual CEntity *Create();
};

class CFactoriaEntidades {
public:

    CEntity *Create(EntityType type);
    void Register(CConstructorEntidad *constructor,
                  EntityType type);
    void Unregister(CConstructorEntidad *constructor);

private:
    std::map<EntityType, CConstructorEntidad*> constructores;

};
```

Figura 2.6: Código C++ de factoría extensible

La implementación hace uso de una tabla hash o diccionario, cuya clave es el identificador o nombre de la clase entidad, y cuyo valor es el método responsable de la verdadera creación del objeto.

Un ejemplo de implementación, basado en el de Llopis (2005b) aparece en la figura 2.6. Existe una clase que hereda de `CConstructorEntidad` por cada uno de los tipos de entidades disponibles, cada una de ellas creando un objeto de esa clase de la jerarquía. La factoría permite registrar y deregistrar los constructores en tiempo de ejecución. Cuando el cargador del mapa detecta que es necesaria la creación de una nueva entidad de un tipo dado, utiliza la factoría, que le devolverá un puntero a la clase base de todas las entidades.

En el ejemplo, hemos suprimido intencionadamente la definición del tipo `EntityType`. En la figura 2.5, se definía como un tipo enumerado. En realidad, se puede utilizar directamente como identificador un entero único para cada tipo de entidad, o una cadena, de tal forma que no sea necesario "centralizar" en un único sitio del proyecto toda la colección de identificadores.

Para facilitar el registro automático de todos los tipos de entidades disponibles en el juego, Llopis (2005b) propone hacer uso de la inicialización estática de C++, aunque expertos del lenguaje desaconsejan el uso de esta técnica (Alexandrescu, 2001).

Gracias a la factoría extensible se evita el problema de tener fijado de antemano en el propio código de la factoría los tipos de entidades posibles. Sin embargo, no resuelve el segundo de los problemas comentados anteriormente.

El cargador del mapa seguirá esperando la creación de un objeto que añadirá a la lista de entidades creadas. Es posible, como comentábamos, que lo que en el mapa viene representado como "entidad", no requiera un tratamiento especial dentro del código, debido a que sea una simple marca o información a añadir a un subsistema (como puede ser un vértice en el grafo de búsqueda de caminos). En ese caso, el "constructor" de la entidad que teníamos antes debería simplemente leer la posición del mapa, y añadir un vértice/punto al grafo de entidades global.

Para poder combinar las dos cosas, una solución es alterar la factoría extensible para que lo que llamábamos `CConstructorEntidad`, en el momento de la creación *no* tenga que devolver una entidad creada a su invocador, sino simplemente si se pudo gestionar la creación o no. En caso de que el constructor estime que debe crearse una entidad (objeto de una clase derivada de `CEntity`), registrará esa entidad invocando al sistema gestor de entidades.

Otra alternativa no orientada objetos es hacer que ese "constructor" no sea un miembro de una clase, sino una mera función C. En ese caso, se puede incluso suprimir la existencia de la factoría, haciendo uso de la tabla de símbolos del ejecutable proporcionada por el sistema operativo (Richter, 1997; Bilas, 2000). El juego Half-Life 1 (Valve Software, 1998), hacía uso de este mecanismo; el motor del juego cargaba una librería dinámica (DLL) que contenía toda la parte específica del juego. Esa DLL exportaba una función por cada uno de los distintos tipos de entidades que podían encontrarse en el mapa. El cargador del nivel en el motor del juego, al encontrar una entidad, buscaba la misma función con ese nombre, haciendo uso del API del sistema operativo subyacente (en este caso Windows).

El último paso para hacer por completo independiente el cargador (o intérprete) del fichero de mapas contenido en el motor del juego y el propio código del juego es hacer que la construcción de las entidades esté en un lenguaje de script (ver apartado 2.7); así está hecho por ejemplo en Far Cry (Crytek Studios, 2004), cuya arquitectura está basada fundamentalmente en el uso de LUA para la implementación de los comportamientos específicos del juego, dejando a C++ la parte del motor.

2.6.3. Arquitectura basada en componentes

El diseño de las entidades basándose en el mecanismo de herencia es muy intuitivo. Sin embargo, existen una serie de problemas que están provocando un cambio en su forma de programación:

- Descomposición: existen infinidad de formas de descomponer el sistema de objetos de un juego, utilizando distintas clasificaciones. El problema, ya de por sí complicado, de decidir qué clasificación es más adecuada se ve acrecentado con la naturaleza cambiante del juego. Los diseñadores de contenido, según avanza el desarrollo, van tomando decisiones que,

de haber sido conocidas al principio, habrían llevado a estructurar la jerarquía de otra forma.

- Código poco flexible: enlazado con los comentarios anteriores, una jerarquía de clases es muy complicada de cambiar. Por lo tanto, ante apariciones de nuevos tipos de entidades no planificados, la solución adoptada suele ser el uso de una sola clase para entidades parecidas pero distintas, la sobrecarga de métodos que un buen diseño nunca habría permitido, etc.

- Clases base grandes: según van creciendo las clases (y entidades) del juego, la jerarquía se hace más grande. En muchas ocasiones, los programadores se encuentran con que la funcionalidad que necesitan para una clase profunda en la jerarquía, ya ha sido implementada en otra clase hermana. Pensemos, por ejemplo, en la jerarquía parcial del Half-Life 1 presentada en la figura 2.4: según la jerarquía, no parece fácil conseguir que una puerta (`CBaseDoor`) pueda romperse (`CBreakable`). Para conseguirlo, lo más fácil es *copiar* el código de la última en la primera, para proporcionarle esas características. Dado que la duplicación no es conveniente, para evitarla, la tendencia natural es *subir* esa funcionalidad a la primera clase padre común. Con el tiempo, eso provoca que las clases superiores de la jerarquía se llenen de métodos que no son utilizados por todas las clases descendientes, sino sólo por algunas. Como ejemplo, podemos nombrar la clase base de las entidades en el Half-Life 1, que tenía 87 métodos y 20 atributos públicos, y la de los Sims que terminó con más de 100 funciones (Brock, 2006). A pesar de que el programador intente estructurar los ficheros de definición por subsecciones agrupando funciones relacionadas, el resultado sigue siendo ficheros grandes difíciles de manejar.

- Clases difíciles de entender: para conseguir entender el comportamiento de una clase, el programador tiene irremediablemente que comprender primero las clases de toda la línea sucesoria desde la clase base.

- Extensión del comportamiento: en una jerarquía como la de las entidades, es muy habitual que las clases derivadas no se limiten a *sobreescribir* el comportamiento de una serie de métodos heredados, sino que lo *extiendan*. Eso, irremediablemente, pasa por *invocar* en algún momento de la implementación del método al mismo método de la clase padre. Dado que el lenguaje no proporciona ningún mecanismo para obligar al programador de la clase hija llamar recursivamente a la implementación de la padre, se presta mucho a olvidos únicamente detectables a través de la depuración. Una solución posible es utilizar el patrón Template Method (Gamma et al., 1995), pero en la práctica, dado que la jerarquía suele ser profunda y que el número de métodos por clase

que lo requerirían es grande, utilizarlo en todos ellos supondría una sobrecarga de nombres difícil de justificar.

- Aparición de herencia en diamante: la jerarquía de clases al fin y al cabo es una clasificación de las entidades utilizando un cierto criterio. En ocasiones surgen entidades que, por su naturaleza, deben formar parte de dos o más categorías. La implementación obvia es hacer que la nueva clase herede de *las dos* categorías, implementando la herencia en diamante. Para implementarla, se utiliza la herencia *virtual* disponible en C++ pero que es en general mal entendida por los programadores, introduce ineficiencias (Lippman, 1996) y acarrea problemas relacionados con la herencia múltiple (Meyers, 1997, Item 43).

- Mayor tiempo de compilación: la herencia es uno de los mecanismos de C++ que provocan mayor acoplamiento entre clases, no sólo desde el punto de vista lógico, sino también físico. Este último tipo de acoplamiento es el que obliga al compilador a tener que procesar todos los ficheros de cabecera de las clases padre, y al enlazador acceder a todos los ficheros objeto para resolver las dependencias. Eso provoca un aumento significativo del tiempo de generación del ejecutable final (Lakos, 1996).

Una solución es dividir la entidad en un conjunto de funcionalidades independientes que se separan en distintas clases. La entidad u objeto del juego pasa así a ser un mero *contenedor de punteros* a esas clases, como por ejemplos las que gestionan el objeto gráfico de la entidad, el objeto físico, el emisor de sonido o el controlador de su lógica (o inteligencia artificial). Para poder comunicar cada una de estas clases, existen dos alternativas principales:

- Que cada una de las clases que implementan la funcionalidad pueda tener un puntero al resto de clases con las que se tiene que comunicar. Por ejemplo, la inteligencia artificial establecerá distintas fuerzas en el objeto físico, por lo tanto tendrá internamente un puntero a él. De acuerdo con la simulación física, ésta actualizará la posición en la que el objeto gráfico tendrá que dibujar el modelo, por lo que el primero tendrá un puntero al segundo.

- Utilizar un sistema de mensajes genérico, de tal forma que uno de los objetos pueda enviar mensajes al resto de ellos informando, por ejemplo, de un cambio en la posición de la entidad.

La creación de una entidad pasa a convertirse en la creación e inicialización de los objetos que implementan las distintas funcionalidades que forman la entidad. Si una entidad no necesita alguna de la funcionalidad (como por ejemplo el emisor de sonido), ese puntero apuntará a `NULL`.

La composición viene a evitar un problema asociado con la centralización de todos los objetos del juego en una única clase. Como ya hemos dicho en la sección 2.3.1, hoy día la industria del videojuego está basada en componentes (o librerías) comprados a terceros. Dado que la gran mayoría de estas librerías son utilizadas desde las entidades, sus programadores necesitan tener un gran dominio de las mismas. La descomposición de la entidad en funcionalidades disjuntas permite también la especialización de los programadores, de forma que únicamente los responsables de la programación de una funcionalidad determinada deberá conocer a la perfección la librería correspondiente.

La generalización del método de composición consiste en hacer que todas las clases que implementan las distintas funcionalidades, *hereden* de una misma clase común, que representa un *componente* genérico. La clase que representa la entidad, en vez de tener una serie de punteros como antes, es un *contenedor de componentes genéricos* (West, 2006). La definición de una entidad consiste en listar de qué componentes estará formada, y cómo configurarlos. La ventaja adicional es que esos componentes se pueden enganchar y desenganchar de la entidad dinámicamente, dependiendo del momento del juego. Por ejemplo, una entidad que representa un vehículo contiene los componentes necesarios para representar una simple entidad estática, que pasa a representar al jugador (cambiando todos los componentes necesarios) cuando su avatar se sube en él.

El modelo de componentes tiene dos ventajas adicionales: permite almacenar todos los componentes del mismo tipo consecutivos en memoria, lo que agiliza sus recorridos, y la división hace que sea más fácil manejar diferentes hebras, cada una encargándose de una funcionalidad particular.

El estilo de agregación *puro* es llevar la idea aún más lejos, y eliminar la necesidad de una clase que represente la entidad. En la figura 2.7, aparece la estructura en memoria de uno de estos sistemas. Cada entidad se convierte en la suma de los componentes cuyo identificador coincide. Existe un gestor para cada uno de los tipos de componente, que guarda una lista con cada uno de esos componentes, para poder actualizarlos en cada vuelta del bucle.

El modelo de componentes "puro" permite tener las entidades almacenadas en una "base de datos"[10]. Cada entidad está distribuida en varias tablas, una para guardar la información relacionada con las colisiones, otra con los componentes de IA, una tabla para los modelos, etc.

Para terminar, el uso de un modelo basado en componentes permite la creación de las entidades dirigida por datos. Cada tipo de entidad viene definido por la lista de componentes que son utilizados por ella, parametrizados con sus valores de inicialización. Eso permite almacenar en bases de

[10]Por ejemplo, algunos de los juegos de Electronic Arts ya llevan integrado un sistema de base de datos (Brock, 2006), y el framework Mangalore montado sobre Nebula 2 utiliza SQLlite para guardar las entidades.

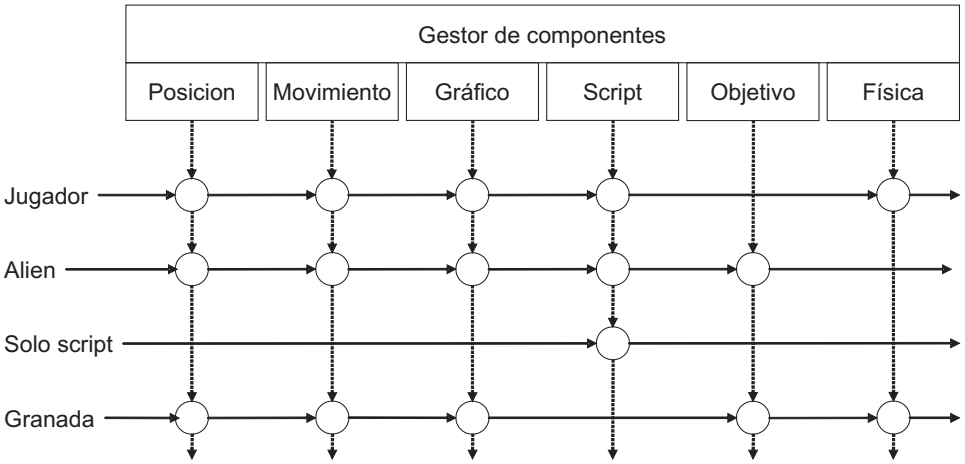


Figura 2.7: Estructura de objetos utilizando componentes (West, 2006)

datos o ficheros XML (*eXtensible Markup Language*, Lenguaje extensible de marcado) la mayor parte de los datos de creación necesarios (Garcés, 2006).

2.7. Lenguajes de script

La evolución de los juegos dirigidos por datos que guardan en ficheros todos los recursos gráficos fue sacar fuera del ejecutable parte del comportamiento del juego. Ya hemos visto en el apartado 2.4 que una alternativa es dividir el juego en dos partes, el *motor de juego* y el *código específico*. Fijando una comunicación entre ambos componentes, se consigue separarlos de tal manera que la parte específica puede ir en una librería de vinculación dinámica (DLL). De esta forma, el equipo de desarrollo puede estructurarse más fácilmente, se reducen las *restricciones físicas* entre partes del código fuente (Lakos, 1996), lo que reduce los tiempos de compilación y enlazado, y se permite la posterior modificación de parte del juego, ya sea por parte del propio estudio de desarrollo, o por aficionados.

El uso de lenguajes de script va un paso más allá. En vez de sacar el código específico del juego utilizando librerías dinámicas, se saca *el propio código*. El motor del juego se convierte así en un *intérprete* del lenguaje de script particular, y lo invoca cuando lo necesita.

Dependiendo del juego, el uso del lenguaje de script será más o menos anecdótico. Por ejemplo, algunos juegos lo pueden utilizar únicamente para el control de escenas entre niveles o lanzamiento de algún comportamiento predefinido, mientras que otros implementan toda su lógica y comportamiento utilizándolos.

La decisión de dónde poner la frontera entre el motor del juego y los

scripts es importante, y afecta al API resultante entre ambas partes y al rendimiento del juego. Como veremos en el apartado 2.7.1, una de las desventajas de los lenguajes de scripts es que su ejecución es más lenta que la del código compilado. Por lo tanto, si se delega mucha funcionalidad en el lenguaje de script, se corre el riesgo de ralentizar demasiado el juego. Si, por el contrario, se cede muy poco control al script, se van perdiendo todas las ventajas, ya que muy poco código podrá aprovecharse ellas.

Desde el punto de vista arquitectónico, el código que maneja un personaje no jugador (en inglés *non-player character*, NPC) era lo último que quedaba dentro del ejecutable, al haber sacado fuera tanto su modelo y animaciones como las texturas. Por lo tanto, con este último paso se consigue una entidad completamente dirigida por datos, que puede empaquetarse en un único fichero y distribuirlo independientemente del juego. Es la idea que utilizan, por ejemplo en Los Sims, donde los objetos que aparecen en las extensiones no son más que ficheros de datos que contienen tanto los modelos de esos objetos como su comportamiento.

2.7.1. Ventajas y desventajas de los lenguajes de script

Una de las principales ventajas de los lenguajes de script es que se reduce significativamente el tiempo del ciclo de desarrollo editar - compilar - enlazar - ejecutar. Esto es debido a que normalmente un cambio de un fichero de script no requiere la compilación del código del juego y su enlazado con todas las librerías, sino que basta con un simple volcado a disco del cambio. De esta forma, el ciclo que puede durar más de dos minutos en el primer caso se reduce a unos segundos en el último. Algunos motores de juegos están incluso preparados para *recargar* el script "al vuelo" (o "*en caliente*"), de tal forma que no es necesario parar el juego y volverlo a lanzar.

Esta reducción en el tiempo de ciclo es posible debido a que normalmente el lenguaje de script es *interpretado*, es decir, durante la ejecución del juego, el fichero se analiza y se ejecutan sus instrucciones interpretándolas una a una. Eso trae consigo una de las desventajas principales de este tipo de lenguajes: su ejecución es mucho más lenta que la un lenguaje *compilado*. En general, un lenguaje de script interpretado no puede competir en rendimiento con el binario creado a partir del código en C++. Por eso debe elegirse con cuidado qué partes del juego van a ser programadas utilizando scripts; normalmente serán aquellas que definen el comportamiento de alto nivel, y que por lo tanto no necesitan ejecutarse a menudo[11].

Otra causa del menor rendimiento del código de scripts es que los lenguajes poseen características avanzadas que suponen un consumo de CPU para su control. Por ejemplo, la recolección de basura, muy habitual en este

[11] Con "*a menudo*", nos estamos refiriendo a una frecuencia de una o más veces por fotograma.

tipo de lenguajes, hace que el motor de scripts dedique parte de su tiempo a su gestión. Sin embargo, estas características avanzadas disponibles en los lenguajes de script y que no están en C++ son también una ventaja, ya que facilitan el desarrollo, reduciendo significativamente la duración del proyecto, y por tanto, pueden ser un punto decisivo a la hora de seleccionar uno u otro lenguaje.

Si el desarrollo con lenguajes de scripts es lo suficientemente fácil, aparece otra ventaja más: los propios diseñadores o creadores de contenidos del juego pueden modificarlos para cambiar parte del comportamiento.

2.7.2. Ejemplos de lenguajes de script

Existen muchos lenguajes de script, desde los utilizados por los intérpretes de comandos (o *shells*), como `sh` o `csh` a los utilizados en la creación de páginas Web como JavaScript, pasando por los que sirven para procesado de textos, como awk o Perl.

En el área que tratamos, sin embargo, de todo el conjunto de lenguajes nos interesan únicamente aquellos que pueden ser incrustados en aplicaciones huésped hechas en C/C++. El programador de la aplicación tiene a su disposición un *motor de scripts* que es capaz de interpretar ese lenguaje. Además, es posible hacer pública parte de la funcionalidad implementada en la aplicación en el lenguaje de script y viceversa. De esa forma, el código del script puede llamar a funciones implementadas en C++ y viceversa.

Aún con estas condiciones, el número de lenguajes disponibles es elevado. A la hora de decantarse por uno u otro lenguaje, hay que tener en cuenta, al menos las siguientes cuestiones:

- Características del lenguaje: dependiendo del uso que se le vaya a dar, se buscarán unas u otras. Habrá que plantearse si se quiere un lenguaje de propósito general que sustituya a C++ en la implementación del código específico del juego, o uno con características especializadas para, por ejemplo, crear el interfaz del usuario o las animaciones de los personajes. En cualquier caso, también es posible que el juego integre distintos motores de scripts para poder soportar más de un lenguaje.

- Rendimiento: también ligado al uso que se le vaya a dar al lenguaje, se podrá pagar más o menos precio en el rendimiento en tiempo de ejecución. Si los scripts se van a utilizar únicamente para el sistema de menús, se podrá optar por un lenguaje más lento pero con características más avanzadas que lo hagan una mejor herramienta de programación. En general, el número de características que tiene un lenguaje es inversamente proporcional con su rendimiento.

- Plataforma: no todos los lenguajes de scripts están disponibles para todas las plataformas. Si un estudio desarrolla para una plataforma es-

pecífica, tendrá que asegurarse que el lenguaje de script está disponible para ella; esto es especialmente importante cuando se desarrolla para consolas.

- Soporte: hoy por hoy los entornos integrados de desarrollo (IDE) para C++ son los suficientemente avanzados como para permitir una programación y depuración cómoda. Sin embargo, esto puede no ser cierto en el lenguaje de script elegido. Si el número de líneas que se programarán en él es elevado, hay que tener en cuenta si su escritura y depuración va a resultar fácil o no, y si no lo es (especialmente la depuración), cuánto cuesta crear un sistema que lo haga sencillo.

Hoy por hoy, el lenguaje de Script más utilizado es LUA (Ierusalimschy et al., 2006). El lenguaje fue desarrollado con unos objetivos claros que encajan con los requisitos de un videojuego: simplicidad en el lenguaje, eficiencia en la ejecución, portabilidad y capacidad de ser *incrustado* en otras aplicaciones con bajo coste de memoria (Ierusalimschy et al., 2005). Otras dos características no menos importantes son su licencia, que permite el uso del motor de scripts en cualquier desarrollo ya sea comercial o no, y las corutinas (de Figueiredo et al., 2006).

Desde el principio LUA ha permitido una doble vertiente: el motor de scripts puede interpretar directamente código LUA, o éste puede *compilarse* a una máquina virtual, e interpretar ese código compilado[12]. De esta forma, durante el desarrollo se puede utilizar el código LUA directamente, para reducir el tiempo del ciclo editar - compilar - enlazar - ejecutar, y en la versión final, los scripts pueden distribuirse compilados para mejorar la eficiencia y mantener el código oculto.

La llegada de LUA al mundo de los videojuegos se produjo en 1998, de la mano de Grim Fandango de LucasArts, con el que el estudio abandonó el sistema de scripts SCUMM que llevaban utilizando durante muchos años. Desde él, le han seguido una lista interminable de juegos, entre los que destacan, además de las aventuras desarrolladas posteriormente por LucasArts, otros juegos como Baldur's Gate de Bioware o Far Cry de Crytek.

Otro lenguaje de propósito general utilizado en algunos juegos es Python (van Rossum, 2006). Su sintáxis clara y sencilla, similar a la de C hacen de él un lenguaje fácil de aprender. El número de librerías disponibles hacen de Python un lenguaje potente y funcional, a lo que se le suma sus capacidades de orientación a objetos, su licencia y su amplio soporte tanto en documentación y tutoriales como en herramientas de depuración. Su principal desventaja es su alto consumo de memoria, sobre todo teniendo en cuenta los entornos limitados que suponen las consolas. Por eso, la mayoría

[12] Desde la versión 5.0 de LUA, esa máquina virtual ha sufrido cambios significativos, y se ha convertido en una máquina basada en registros, en vez de basada en una pila de operandos, por lo que, aseguran sus autores, la ejecución del código LUA es más rápida.

Figura 2.8: Captura del juego Don Quijote (Dinamic, 1987)

de los juegos que utilizan Python están disponibles únicamente en PC, como el Civilization IV.

Por último, nombraremos el caso de Java, un lenguaje de propósito general que, no obstante, puede utilizarse como lenguaje de script. En el apartado 2.7.3.4 veremos un ejemplo concreto de juego que lo hace.

Si ninguno de los lenguajes de script disponibles se amolda a las necesidades del estudio, la última alternativa es desarrollar uno propio. Esta decisión hay que tomarla con cautela, ya que la implementación de un analizador e intérprete eficiente entra dentro del área de los compiladores (de hecho, es una de las razones que llevaron a LucasArts a utilizar LUA, una vez decidieron abandonar SCUMM). Aún así, hay varios juegos que utilizan lenguajes especialmente diseñados para ellos. La ventaja de esta aproximación es que se puede crear un lenguaje dedicado, con características que no se encuentran en los lenguajes de propósito general, pero que facilitan la creación de los scripts del juego. Veremos un ejemplo de esto en el apartado 2.7.3.3.

2.7.3. Casos de uso

Los siguientes apartados describen algunos de los casos más significativos del uso de lenguajes de scripts en juegos.

2.7.3.1. Zork

Zork fue uno de los primeros juegos de lo que después se vino a llamar "*ficción interactiva*", por lo que se le puede considerar el padre o abuelo de los llamados "juegos conversacionales" (como el Don Quijote (Dinamic, 1987), que puede verse en la figura 2.8) y de las aventuras gráficas (como Grim Fandango, ver apartado 2.7.3.2). En sus orígenes, estos juegos se basaban en una descripción textual de una situación, y una interacción a base de texto

(órdenes muy sencillas), para moverse y operar sobre el entorno.

Lo peculiar de este juego es el modo en que estaba implementado. Las primeras versiones (allá por 1971), hacían uso de lenguajes de implementación parecidos a Lisp. Con el incremento por parte del público de su interés, sus creadores formaron una empresa para explotar el éxito, que convirtió a Zork en una *serie* con numerosas aventuras. En 1979, desarrollaron una *máquina virtual*, llamada *máquina-Z*, creada expresamente para facilitar la programación de aventuras textuales.

La máquina virtual interpretaba instrucciones extraídas del *fichero de historia*, o fichero-Z, independiente de la plataforma en la que se ejecutaba la máquina virtual. De esta forma, se convirtieron en los primeros en separar la lógica del juego (contenida en esos ficheros) del propio motor, creando juegos dirigidos por datos. Eso les permitió llegar a un gran número de plataformas, simplemente implementando la máquina-Z para todas ellas.

Para facilitar la programación de cada una de las historias interactivas, utilizaron un lenguaje de más alto nivel que el interpretado por la máquina virtual, conocido como ZIL (Infocom, 1989), acrónimo de *Zork Implementation Language*. Si bien el compilador nunca fue liberado por la compañía, posteriormente un amante de las historias interactivas diseñó e implementó un lenguaje distinto llamado Inform (Nelson, 2001), cuyo compilador genera instrucciones para la máquina-Z, haciendo posible aún hoy escribir aventuras textuales nuevas utilizando un lenguaje de alto nivel.

Por poner un ejemplo concreto de un juego de ficción interactiva, podemos pensar en la forma de implementar el juego español Don Quijote (Dinamic, 1987) desarrollado para distintos ordenadores de 8 bits. La figura 2.8 muestra una captura de un momento del juego. Una alternativa de programación sería incrustar toda la lógica en el propio código, mientras que la otra es utilizar una máquina-Z o un intérprete de un lenguaje de script especialmente diseñado para la ficción interactiva, que permita definir toda la lógica de manera independiente al motor del juego que la lee, analiza e interpreta. La figura 2.9 muestra cómo podría ser el código de ambas alternativas: la primera de ellas en BASIC, y la segunda utilizando Inform 6 (Nelson, 2001).

2.7.3.2. Grim Fandango

Grim Fandango (LucasArts, 1998) de LucasArts fue el primer juego en utilizar LUA. Después de usar su sistema de scripts propietario llamado SCUMM desde 1988, la primera idea en el desarrollo de Grim Fandango era extender el sistema para soportar personajes en 3D.

Al principio del desarrollo, se hizo evidente que el paso a las tres dimensiones era demasiado complicado utilizando la base de código antigua, y Bret Mogilefsky, el programador jefe, decidió empezar un nuevo motor de

```
1000 REM Sala de la biblioteca
1010 GOSUB 1200 ' Dibujar la biblioteca
1010 PRINT "Los libros polvorientos destacan en una"
1020 PRINT "estanteria situada hacia a la izquierda."
1030 PRINT "Una puerta abre el camino hacia el sur."
1040 PRINT "Tambien puedes observar que hay un"
1050 PRINT "libro"
1060 GOSUB 5000 ' Leer entrada
1070 IF entrada$="INVENTARIO" OR entrada$="I" OR
entrada$="INV" THEN GOSUB 6000: GOTO 1000
1080 IF entrada$="LEER LIBRO" AND NOT libroYaLeido THEN
PRINT "Empieza la aventura":libroYaLeido = 1: GOTO 1000
1090 IF entrada$="IR AL SUR" OR entrada$="SUR" GOTO 1500
```

a) Fragmento de código en BASIC

```
! Descripcion de la biblioteca
Room  biblioteca "Biblioteca de Alonso Quijada"
   with description [;
print "Los libros polvorientos destacan en una estanteria
       situada hacia a la izquierda. Una puerta abre el
       camino hacia el sur.Tambien puedes observar que hay
       un libro";
        ]
        s_to habitacionAlonsoQuijada
[.....]
! Descripción de la habitación
Room habitacionAlonsoQuijada "Habitación de Alonso Quijada"
   with description [;
[.....]
```

b) Fragmento de código en Inform

Figura 2.9: Código posible para el juego Don Quijote

aventuras gráficas completamente nuevo. Fue así como se vio enfrentado a la decisión de elegir un nuevo lenguaje de scripts para manejar la funcionalidad del motor.

El resultado final fue un nuevo sistema para crear aventuras gráficas, llamado GrimE. En la práctica, GrimE no sabe nada de las dinámicas que se utilizan en las aventuras gráficas, ya que toda su lógica está fuera del motor, programada en LUA. El motor únicamente proporciona primitivas para cargar escenarios, personajes, escribir texto y gráficos primitivos, y alguna otra función relacionada con la reproducción de animaciones y búsqueda de caminos. Por poner algún ejemplo, algunas funciones que se hacen públicas en LUA son `TurnActorTo`, `StopActorChore` o `CanActorSee`.

Por lo tanto, como el propio Bret reconoce (Mogilefsky, 1999), gran parte

del juego está hecho en LUA. Haciendo un análisis utilizando una versión modificada de Residual[13], de los 14.894 ficheros empaquetados en 1'2Gb de datos que se utilizan en Grim Fandango, 783 corresponden con guiones LUA (compilados), que suman algo más de 2 megabytes.

Hay que hacer notar que, al contrario de lo que ocurre con otros lenguajes de script, la evolución de LUA está en ocasiones guiada por los propios desarrolladores de juegos (Ierusalimschy et al., 2007). Cuando se inició el proyecto de Grim Fandango, LUA estaba en la versión 2.5, y carecía de muchas funcionalidades deseables para una programación de juegos cómoda. Una de esas características era el nulo soporte para la multitarea. Es por esto que el equipo de desarrollo extendió el intérprete para soportar una especie de multitarea colaborativa que permitía que cada personaje tuviera una especie de hebra que ejecutaba su comportamiento. Durante el propio desarrollo, y ante un cambio de versión en LUA, este cambio lo migraron a la versión 3.1. Estas extensiones que fueron haciendo en paralelo varios estudios y permanecían bajo secreto industrial era ampliamente demandada por toda la comunidad de usuarios de LUA, y finalmente en la versión 5.0 de 2003 se añadió la capacidad de gestionar *corutinas* de manera nativa en el lenguaje, como un método de multitarea cooperativa. La versión 5.1 también incorporó otra característica demandada por los desarrolladores de juegos: un recolector de basura incremental, que evita las paradas temporales de todo el motor de script que se producían cuando éste empezaba a actuar.

2.7.3.3. Unreal Tournament

Unreal Tournament (Epic Games, 1999) es una serie de juegos FPS que comenzó en 1999, y que aún sigue en la actualidad. El motor gráfico que utilizan es uno de sus puntos fuertes[14], al que se le suma una inteligencia artificial sofisticada.

El juego permite tanto el modo de un solo jugador como el multijugador en red. Además, su estructura facilita a los aficionados la modificación del juego, creando nuevos escenarios, objetos y personajes.

La característica más destacable es que el comportamiento de estos personajes está programado utilizando un lenguaje de scripts. En este caso, después de un análisis cuidadoso de los lenguajes existentes (en el que analizaron por ejemplo Java y alguna versión de VisualBasic), el equipo de desarrollo terminó creando un lenguaje propio, conocido como *UnrealScript* (Sweeney et al., 2004).

El lenguaje opera a alto nivel, y soporta de forma nativa el tiempo y la red. Igual que en Java, no utiliza punteros, tiene recolección automática

[13] `http://www.scummvm.org`

[14] De hecho, Epic, la empresa que desarrolla el juego, licencia el motor a otros estudios.

de basura, no soporta herencia múltiple y es fuertemente tipado. También existe una clase base de todas las clases, llamada `Object`.

Lo más destacable del lenguaje es la forma en que está diseñado para una fácil integración en el proceso de desarrollo del juego, y su característica para facilitar la programación de máquinas de estados.

Con respecto a lo primero, cabe mencionar tres aspectos:

- En la propia definición de las clases, que implementan el comportamiento de una entidad/NPC, se pueden marcar qué atributos podrán modificarse o establecerse desde el editor de niveles del juego (UnrealEd). Cuando el diseñador del nivel coloca una entidad, el editor mira el código fuente asociado (programado en UnrealScript), y presenta un interfaz construido expresamente para esa clase, que permite cambiar todos sus atributos publicados al editor.

- Además de las clases abstractas y nativas (aquellas cuyos métodos son implementados por el motor en C++), existe un tipo especial de clase que no está disponible en lenguajes de propósito general, llamadas "clases temporales". Estas clases son gestionadas por el módulo de persistencia del motor de manera especial, de tal forma que cuando hay que llevar a disco el estado del juego, las ignora.

- Los atributos `const` declarados en las clases no indican, pese a lo que podría parecer, que esos atributos no cambian a lo largo de la vida del objeto, sino que *desde el código del UnrealScript* el atributo no cambia, es decir que son de sólo lectura. Su valor es actualizado por el motor del juego; por lo tanto, su significado es similar a lo que sería en C++ una variable con los modificadores `volatile const`.

No obstante, la característica más importante del lenguaje, que no aparece en los lenguajes de programación de propósito general, es su soporte nativo para la programación de máquinas de estados. Las máquinas de estados son una de las técnicas preferidas para la creación de comportamientos de los NPC (Dybsand, 2002; Houlette y Fu, 2004; Rosado, 2004), por lo que el soporte nativo representa una ventaja.

Cada clase de UnrealScript que representa a una entidad, implementa una serie de métodos que determinan las acciones que la entidad realiza ante unos determinados eventos. Lo normal a la hora de programar la IA es que ese comportamiento dependa del estado del NPC (si se está atacando o vagando por el entorno, etc.). En vez de utilizar la instrucción `switch` en todos esos métodos, UnrealScript permite en cada clase la definición de estados.

Un estado no es más que una agrupación de métodos que implementan las acciones a realizar en cada evento, de tal forma que un objeto puede tener activo un único estado. El lenguaje permite herencia entre estados, igual que

```
void CEntity::FindCover() {
    vector3 nearCover;

    nearCover = findNearCover(getPosition());

    moveTo(nearCover);
}
```

Figura 2.10: Pseudo-código de un comportamiento en busca de cobertura

permite herencia entre clases normales, lo que facilita mucho su programación, evitando repetición de código en estados que definen comportamientos parecidos.

Una ventaja fundamental de tener soporte nativo de estados y máquinas de estados en el propio lenguaje es que la invocación de los métodos es consciente de la existencia de estos estados. De esta forma, cuando el motor invoca un método de una entidad, para informar sobre la ocurrencia de un cierto evento, el mecanismo de invocación realiza las siguientes acciones:

- Si el objeto (entidad invocada) tiene activo un estado actual:
  - Si el estado tiene un método con ese nombre, se invoca,
  - si no, si algún antecesor del estado (clases padre de la clase que define el estado) tiene un método con ese nombre, se invoca el más específico.
- Si el objeto no tiene un estado actual, o la fase anterior no encontró ningún método válido:
  - Si la clase tiene un método con ese nombre, se invoca,
  - si no, se busca en las clase padre, utilizando el mecanismo tradicional en orientación a objetos.

Por último, soluciona de forma elegante un problema común cuando se utilizan scripts en lenguajes de programación: la ejecución de funciones que duran más de un *frame* (acciones "no instantáneas"). El problema se da, por ejemplo, cuando queremos hacer que una entidad se esconda detrás de una *cobertura*. El pseudo-código sería parecido al que muestra la figura 2.10. La función `moveTo` recibe la posición a la que se debe mover la entidad, calcula la ruta para salvar los obstáculos y la hace andar por ella hasta el final.

Dado que la función utiliza varios ciclos de juego (o vueltas al bucle principal) para ejecutarse, el método bloquearía por completo el sistema. En UnrealScript han solucionado el problema por medio de lo que llaman *funciones latentes*, que son funciones especiales que hace públicas el motor y

cuya ejecución puede requerir más de un *frame*. El motor de scripts *bloquea* automáticamente la ejecución de la función, almacenando internamente su estado, mientras el motor ejecuta la función latente `moveTo`. Cuando ésta termina su ejecución, automáticamente el motor de scripts reanuda la ejecución del método. De esta forma, el código del script puede tener una sucesión de funciones latentes fáciles de entender y programar.

2.7.3.4. Vampire Masquerade Redemption

La peculiaridad del Vampire (Nihilistic-Software, 2000) es que su sistema de scripts está implementado en Java, un lenguaje que en principio no tiene las características más adecuadas para ser utilizado como lenguaje de script.

El equipo de desarrollo después de haber terminado otro proyecto en donde implementaron su propio lenguaje de script, decidieron no caer en el mismo error de perder meses de desarrolladores programando y arreglando errores del sistema de scripts, por lo que hicieron un análisis de los lenguajes existentes y se decantaron por Java. El resultado final fue un motor de juego con unas 300.000 líneas de código, y un conjunto de scripts en Java que sumaban otras 66.000 (Huebner, 2000).

Para utilizar Java como lenguaje de script, aprovecharon el interfaz estándar de Sun que permite la comunicación entre aplicaciones de Java y librerías nativas, llamado JNI (Liang, 1999). Aunque el cometido original de JNI (*Java Native Interface*) era permitir que las aplicaciones Java llamaran a librerías desarrolladas en otro lenguaje, en este caso, el uso que le dieron fue el inverso: llamar a Java desde el motor en C++[15].

Según Huebner (1999), cada una de las entidades controladas por el sistema de script estaba implementada como una clase Java independiente. El motor, al leer los datos del mapa y detectar las entidades, iba cargando cada una de esas clases. Utilizando el sistema de *introspección* o RTTI de Java (Eckel, 2003), detectan a qué tipos de eventos es capaz de reaccionar (mirando qué métodos implementa).

El mismo RTTI es utilizado por el propio editor de niveles, para presentar al diseñador las opciones precisas dependiendo del tipo de entidad que está editando. No obstante, en ese caso, dado que RTTI no permite extraer el *nombre* de los parámetros de los métodos (ya que se pierde en la compilación), para poder indicar las cadenas que el editor presenta al diseñador, permiten al programador de la clase la creación de atributos estáticos bajo unas determinadas condiciones, que el editor leerá para presentar al usuario. Por ejemplo, el editor de niveles permite al diseñador indicar los parámetros que se pasarán al *constructor de la clase* que gobierna la entidad añadida. Utilizando RTTI, lo único que el editor podría presentar sería el número de

[15]Como veremos en el capítulo 6, nuestro sistema utiliza la misma técnica. Puede encontrarse una explicación detallada en el apéndice B.

parámetros y su tipo. Para mejorarlo, el programador de la clase puede crear un vector de cadenas como atributo estático, en el que se guarda una cadena por cada parámetro, almacenando su descripción y su valor por defecto.

Dado que Java carga las clases cuando se necesitan y activa el recolector de basura cuando queda poca memoria, es posible que el uso de CPU por parte de la máquina virtual presente picos inesperados que haga que el juego sufra parones esporádicos. Para evitar eso, en Vampire forzaban la carga de todas las clases en el momento de la carga del nivel, y utilizaban referencias globales a todos los objetos, de tal forma que el recolector de basura no tuviera nada que liberar en ningún momento. En periodos de baja actividad del juego, liberaban las referencias de los objetos no utilizados, e invocaban manualmente al recolector de basura. De esta forma, minimizaban al máximo su impacto negativo.

2.7.3.5. Far Cry

Far Cry (Crytek Studios, 2004) es un FPS en el que el jugador se ve inmerso en una isla paradisiaca, enfrentándose a numerosos peligros. El juego puede ser jugado tanto por un único usuario, como por varios a través de una red.

La arquitectura de Far Cry está desarrollada pensando en la fácil modificación por parte de aficionados y profesionales, mediante un editor de niveles disponible con la propia versión del juego, y un conjunto de ficheros de código fuente descargables y empaquetados en un kit de desarrollo (SDK).

La característica de Far Cry que nos interesa aquí es que hace un uso intensivo de LUA como lenguaje de script. En particular, el juego utiliza hasta 670 archivos distintos que suman un total de casi 4'6Mb de código fuente[16].

LUA está perfectamente integrado en el editor de niveles, llamado Cry Engine Sandbox (Crytek, 2004). La integración llega hasta tal punto que éste permite editar algunos de los scripts que definen el comportamiento de las entidades, y recargarlos sin tener que reiniciar. Sin embargo, dado que LUA es un lenguaje de script de propósito general y a diferencia de lo que ocurría en el caso del Unreal Tournament, el editor no es capaz de averiguar qué propiedades debe permitir alterar.

La comunicación entre el motor del juego y el motor de scripts está distribuida por los distintos módulos que forman parte del juego. Así, el gestor de red hace públicas sus funciones, el gestor del GUI las suyas, el módulo de Inteligencia Artificial las suyas, etc. Para evitar conflictos de nombres, cada módulo hace públicas sus funciones en un "espacio de nombres" distinto[17].

[16] Esto es especialmente llamativo si lo comparamos con otros datos. Por ejemplo, el código C++ de manejo de todas las entidades de Half-Life 1 Valve Software (1998) consta de 145 archivos, que suman 2'48Mb.

[17] En LUA los espacios de nombres se construyen utilizando tablas.

```
-- Creación del objetivo compuesto cover_look_closer
AI:CreateGoalPipe("cover_look_closer");
AI:PushGoal("cover_look_closer","timeout",1,0.5,1.5);
AI:PushGoal("cover_look_closer","approach",1,0.5);
AI:PushGoal("cover_look_closer","lookat",1,0,-90);
AI:PushGoal("cover_look_closer","timeout",1,0.5);
AI:PushGoal("cover_look_closer","lookat",1,0,90);
```

Figura 2.11: Definición de un objetivo compuesto en Far Cry

En el código LUA de la figura 2.11 se aprecian algunas llamadas a funciones implementadas en el módulo de inteligencia artificial básica del motor del juego.

Para solucionar el problema de la ejecución de funciones que duran más de un fotograma (que en el apartado 2.7.3.3 llamábamos funciones "latentes"), utilizan lo que llaman *objetivos*. El motor del juego soporta unas acciones u objetivos básicos o primitivos, que el programador del script puede combinar para crear un objetivo compuesto.

Por ejemplo, la figura 2.11 muestra el código LUA utilizado en Far Cry para definir el objetivo compuesto `cover_look_closer`, utilizado por los NPCs llamados *cover* cuando pasan a estado de alerta por haber visto u oido algo sospechoso. El comportamiento hace que se acerquen al sonido u objeto percibido, y miren hacia los lados; las funciones a las que se llama para crear el objetivo compuesto (`AI:CreateGoalPipe` y `AI:PushGoal`) están implementadas en el motor del juego (en concreto, en la parte encargada de la IA básica).

Para la programación de los comportamientos de los NPCs, se utilizan máquinas de estados programadas en LUA. Aunque el lenguaje no tiene un soporte nativo para ellas, su capacidad de soportar tablas de manera eficiente y cómoda para el programador, hacen que la creación de cada estado sea relativamente sencillo. El programador únicamente tiene que crear una tabla para cada estado, con una entrada por evento que contenga la función a ejecutar cuando éste ocurre. En la tabla también se puede indicar a qué estado se debe pasar en ese momento.

En la implementación de las funciones que se ejecutan para procesar cada "mensaje" recibido, se puede utilizar toda la potencia de LUA para decidir qué acciones realizar y utilizar llamadas al motor para obtener información adicional. Finalmente, la función puede seleccionar los distintos objetivos (tanto los primitivos como los compuestos), e incluso seleccionar varios, que serán ejecutados en secuencia. La figura 2.12 muestra el código que se ejecuta cuando la entidad *cover* recibe la notificación del módulo de percepción de que ha visto algo distinto al jugador. Como vemos, lo que hace es meter en la pila de objetivos a ejecutar el desenfundar el arma, configurar el modo de

```
OnSomethingSeen = function( self, entity )
-- Evento generado cuando la entidad ve un enemigo
-- distinto del jugador

entity:Readibility("IDLE_TO_INTERESTED");

entity:SelectPipe(0,"cover_look_closer");
entity:InsertSubpipe(0,"setup_stealth");
entity:InsertSubpipe(0,"DRAW_GUN");
end,
```

Figura 2.12: Implementación de un comportamiento en Far Cry

sigilo en todas sus acciones e ir a inspeccionar las inmediaciones del enemigo visto[18].

2.7.3.6. Age of Empires II: The Age of King

El caso del juego de estrategia Age of Emires II (Ensemble-Studios, 1999) es bastante distinto de todos los anteriores. En este caso, el juego hace uso de un *motor de reglas* para la toma de decisiones de la inteligencia artificial controlada por el ordenador. El lenguaje de script utilizado, por tanto, no es un lenguaje imperativo como los anteriores, sino un lenguaje de reglas, que permite la programación *declarativa* del comportamiento.

Por lo tanto, de entre todas las ventajas introducidas por el uso de lenguajes de script, la fundamental en este caso es el motor de inferencia que hay por detrás en la ejecución del lenguaje.

El uso de un sistema de reglas permite un modelo de razonamiento más general, gracias a su carácter declarativo. De esta forma, los comportamientos implementados son de más alto nivel y pueden ser reutilizados en más de un contexto (da Silva y Vasconcelos, 2006; Combs y Ardoint, 2005).

El motor de reglas actúa como un intérprete de instrucciones `if`/`then`. Cada una de esas "instrucciones" es una regla, con unas condiciones que deben satisfacerse para que se ejecuten sus acciones asociadas.

Las condiciones se basan en una serie de hechos que representan el estado del juego, y que pueden entenderse como la percepción de la inteligencia artificial. Los hechos detallan la información sobre el jugador (como la cantidad disponible de un determinado material), sus oponentes (por ejemplo su puntuación) y sobre el propio juego (tiempo transcurrido, condiciones para ganar, etc.).

Para el motor de inferencia, se puede utilizar uno ya existente, como

[18] En el código se aprecia que los objetivos se añaden en *orden inverso* a su ejecución, ya que la estructura de datos en la que se almacenan es una pila.

```
(defrule
  (food-amount > 1700)
  (or
    (wood-amount < 1100)
    (or
      (gold-amount < 1200)
      (stone-amount < 650)))
  (can-sell-commodity food)
=>
  (chat-local-to-self
      "excess food")
  (release-escrow food)
  (sell-commodity food)
)
```

```
(defrule
  (gold-amount > 1250)
  (wood-amount < 1100)
  (can-buy-commodity wood)
  (commodity-buying-price
     wood < 50)
=>
  (chat-local-to-self
     "excess gold; buy wood")
  (release-escrow gold)
  (buy-commodity wood)
)
```

Figura 2.13: Ejemplos del uso de un motor de reglas en un juego

CLIPS (Giarratano, 2002) o Jess (Friedman-Hill, 2005). En el caso concreto de Age of Empires, se creó uno nuevo basándose en los anteriores, con un lenguaje de definición de reglas parecido que soporta algunos comandos útiles para el juego. Si el sistema de inferencia no es demasiado sofisticado, se puede incluso programar en C++ la propia definición de las reglas (Bourg y Seemane, 2004), aunque se pierden las ventajas asociadas a los lenguajes de scripts, como la reducción del ciclo de desarrollo que explicábamos en la sección 2.7.1.

La figura 2.13 muestra dos ejemplos de reglas, ambas relacionadas con los recursos disponibles por el jugador controlado por el sistema de reglas. La primera se dispara para vender comida cuando tiene exceso de ésta pero falta de otros recursos, mientras que la segunda se utiliza para comprar madera cuando se tiene poca, está barata y se dispone de margen de dinero suficiente.

Resumen

En este capítulo se han detallado los componentes más importantes de un juego desde el punto de vista de su arquitectura interna, que más adelante dirigirán el diseño de una arquitectura útil para la creación de sistemas educativos basados en videojuegos.

Después de una breve introducción y una definición de lo que entendemos por arquitectura software, la sección 2.3 expone motivos por los que se hace necesaria la existencia de una arquitectura bien pensada en el desarrollo del software de entretenimiento y, muy posiblemente, la adquisición por parte de los desarrolladores de componentes o librerías externas.

Una vez justificada la existencia de estas arquitecturas, la sección 2.4 presenta a grandes rasgos la característica más importante y común de todos los desarrollos actuales: la extracción fuera del ejecutable final de la máxima cantidad de información posible, o lo que es lo mismo, la idea de las arquitecturas dirigidas por datos.

Tras eso, la sección 2.5 pasa a describir los distintos modelos de bucle principal que hay y la sección 2.6 detalla las posibilidades existentes para la gestión de todos los objetos que hacen del juego un entorno interactivo.

Finalmente, la sección 2.7 recupera la importancia de una arquitectura dirigida por datos. Para eso, detalla las posibilidades que hay para sacar fuera del ejecutable parte del código del propio juego, de forma que se consigue una aplicación fácilmente configurable.

En el capítulo siguiente abandonamos el mundo de los videojuegos, para adentrarnos en los programas educativos. Más adelante, mezclaremos las dos áreas para recuperar el tema original, la creación de aplicaciones educativas basadas en videojuegos.

Notas bibliográficas

En los últimos años, ha habido un incremento significativo en el número de textos relacionados con la creación de videojuegos, dirigidos tanto para un público aficionado como para profesionales del sector.

La mayoría de ellos, no obstante, cubren con detalle un determinado aspecto de la programación de juegos, sin dar ideas de la arquitectura general que se necesita. Especialmente extendidos están las series de libros titulados *Game Programming Gems* (DeLoura, 2000, 2001; Treglia, 2002; Kirmse, 2004; Pallister, 2005; Dickheiser, 2006) y *AI Wisdom* (Ravin, 2002, 2004, 2006). Todos estos libros están formados a partir de artículos independientes, algunos de ellos referenciados a lo largo del capítulo.

Algunos libros intentan dar una visión general de todos los aspectos de la programación de juegos, como Sanchez-Crespo Dalmau (2003) o Rucker (2002), mientras que otros se centran en un único aspecto, como la parte gráfica (Watt, 1999; Moller et al., 2002), inteligencia artificial (Schwab, 2004), física (Kodicek, 2005) o scripts (Varanese, 2002). Por ultimo, cabe destacar Rollings y Morris (2004), que hace un intento de presentar una arquitectura de videojuegos.

Desde un punto de vista más académico, existen algunas conferencias cuyas actas son un buen foco de información. Por ejemplo, existen congresos dedicados a la programación de juegos en general (como la *Game Developers Conference*[19]), a la informática gráfica (ACM SIGGRAPH[20]) y a la

[19] `http://www.gdconf.com`
[20] `http://www.siggraph.org`

inteligencia artificial en juegos (Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE). Este creciente interés ha dado lugar, entre otros, al uso de sistemas de reglas para resolver el buscaminas (Gómez Martín y Díaz Agudo, 2006), la utilización de planificación en coordinación de equipos de enemigos (Muñóz Ávila y Hoang, 2006) o el uso de razonamiento basado en casos para juegos de estrategia (Sánchez Pelegrín, Gómez Martín y Díaz Agudo, 2005).

Por último, para conocer de qué forma están estructurados los distintos juegos, otra alternativa es inspeccionar su código. Existe un gran número de juegos cuyo código ha sido liberado, que pueden utilizarse para ver las distintas aproximaciones que utilizan para resolver los problemas planteados en este capítulo. También es educativo adentrarse en las SDK o kits de desarrollo que ciertos estudios hacen públicos para permitir a jugadores aficionados y empresas realizar modificaciones del juego. Gracias a estos paquetes hemos podido mostrar datos sobre el funcionamiento de Half-Life y de Far Cry.

Capítulo 3

Aplicaciones educativas

El único medio racional de educar es dar ejemplo,
y si no hay otro remedio, un ejemplo que ponga
sobre aviso.

Albert Einstein

En este capítulo hacemos un análisis de las arquitecturas de los programas educativos que centran nuestro foco de atención. Para eso, empezamos por analizar la construcción de sistemas de enseñanza inteligentes. Después analizamos cómo se integran agentes en entornos virtuales para después aplicar esa integración a los agentes pedagógicos.

3.1. Introducción

Desde los primeros años de la informática existió un interés por los sistemas de enseñanza asistidos por ordenador, que a partir de 1970 cristalizan en lo que se llamó Sistemas de enseñanza inteligentes o Sistemas Inteligentes de Tutoría[1] (Carbonell, 1970). Tanto expertos en inteligencia artificial como educadores y psicólogos los vieron como una vía prometedora en la que aplicar sus técnicas y teorías. El primer libro dedicado a ITSs aparece en 1982, de la mano de Sleeman y Brown; después vinieron otros, hoy considerados clásicos, como Kearsley (1987); Lawler y Yazdani (1987); Wenger (1987); Polson y Richardson (1988).

En la construcción de los primeros tutores inteligentes, se hacía hincapié en los conocimientos que el sistema tenía sobre el dominio, técnicas pedagógicas y modelo del usuario, sin tener en cuenta la reutilización de todo ese esfuerzo para el desarrollo de sistemas futuros. Sólo desde hace algún

[1] En el mundo anglosajón son conocidos como *Intelligent Tutoring Systems*, abreviado como *ITSs*, siglas que utilizaremos de ahora en adelante.

tiempo se está prestando atención a la arquitectura general del sistema, a la posibilidad de reutilizar componentes, modelos cognitivos, etc.

En la primera parte de este capítulo haremos una descripción de las características generales que se atribuyen a los programas educativos, para después pasar a analizar las arquitecturas software más comúnmente utilizadas.

3.2. Descripción de los entornos de aprendizaje

Los enfoques de aprendizaje utilizados hoy en día por numerosos educadores tienden a dar al alumno un papel activo a la hora de enseñar nuevos conceptos y procedimientos. Este método se basa en el enfoque constructivista, expuesto inicialmente por Piaget (1955), en el que el énfasis se pone en el papel activo que juega el estudiante en la construcción de su propio conocimiento.

El método del aprendiz o *learning-by-doing* (Schank y Clearyv, 1994; Kolodner, 1997) se basa en este enfoque constructivista. A pesar de que la idea parece novedosa, la realidad es que ha sido la forma en la que tradicionalmente se han transmitido, generación tras generación, las destrezas de muchas profesiones fundamentalmente manuales. El traslado de este tipo de enseñanzas desde el mundo real al ordenador consiste en añadir entornos virtuales de aprendizaje a los sistemas, creando lo que se conoce como *aprendizaje inmersivo*. Durante años, la comunidad persiguió este tipo de aprendizaje a base de la construcción de *micromundos* (ver por ejemplo Burton y Brown (1982); Hollan et al. (1984); Lawler y Lawler (1987); Thompson (1987)), para posteriormente, con el avance de la capacidad de cálculo, explorar los entornos virtuales tridimensionales (Rickel y Johnson, 1997).

Según Bares et al. (1998), el *aprendizaje inmersivo* se caracteriza por:

- **Motivación**: para que el alumno pase tiempo utilizando el programa educativo, debe estar motivado. Básicamente, la motivación puede venir de dos vías (Sansone y Harackiewicz, 2000): (i) los factores externos al estudiante, como las recompensas (motivación *extrínseca*), y (ii) desde el propio estudiante (motivación *intrínseca*). Para conseguir esta última, existen dos factores que el aprendizaje inmersivo favorece: el reto propuesto al estudiante y la curiosidad que siente hacia el problema.

- **Resolución de problemas en situación**: quitando las áreas en las que se ha utilizado el método del aprendiz, en general, las teorías de aprendizaje se han centrado en la memorización de conceptos. La aproximación del aprendizaje inmersivo está dirigida por objetivos más claros: la resolución de problemas concretos. De esta forma, el estudiante va aplicando la teoría a la vez que la aprende.

- **Inmersión con un papel particular**: desde el primer juego de rol (Gygax y Arneson (1974); consultar Kim (2006) para una enumeración exhaustiva), pareció claro que participar activamente en el desarrollo de una historia resultaba especialmente atrayente a los jugadores. Más tarde, los juegos de rol por ordenador (por ejemplo Nihilistic-Software (2000) o BioWare-Corp (2002)) vinieron a confirmar la teoría. Esta misma situación se puede extrapolar a los entornos de aprendizaje, donde el estudiante acepta un papel activo en el mundo virtual que simula el entorno de estudio.

En realidad, para llegar a los sistemas educativos con entornos de aprendizaje basados en el aprendizaje inmersivo y el uso de micromundos, se ha ido pasando por distintos tipos de sistemas más o menos sofisticados.

Los ITSs son la evolución de aplicaciones educativas más sencillas, encuadradas dentro del área de *enseñanza asistida por ordenador* (en inglés "*Computer-Aided Instruction*", abreviado como CAI).

Los sistemas iniciales eran meros contenedores de conocimiento estructurado, igual que lo son los libros. Los autores utilizaban su conocimiento sobre el dominio y su destreza como oradores (o escritores de contenido), para representar los conceptos en la forma apropiada. De la misma forma que en un libro el autor toma ciertas decisiones pedagógicas para secuenciar el contenido en capítulos, o incluso añade invitaciones a lectores avanzados a saltarse secciones, etc., los creadores de contenidos de un sistema CAI decidían el orden de presentación.

A primera vista, lo que distingue a un sistema CAI simple de otro es únicamente el contenido. Por eso, existen herramientas conocidas como generadores de tutores inteligentes, o en inglés *intelligent tutor generators* (Kodaganallur et al., 2006), que ayudan en la creación de esos contenidos, independientemente del área de conocimiento. Estas herramientas han evolucionado desde los antiguos PLATO (Wikipedia (PLATO)) y TICCIT (Alderman, 1979), hasta el moderno Macromedia AuthorWare (Adobe, 2006), sin dejar de lado los esfuerzos en el campo de la investigación como Koedinger et al. (2004). El lector interesado puede consultar Murray (1999) para una amplia revisión de las mismas.

Posteriormente, los sistemas CAI fueron evolucionando. Los más sofisticados empezaron a incluir cierta autonomía en la toma de decisiones, para convertirse en aplicaciones dinámicas, en contraposición a la presentación en un orden preestablecido de los contenidos. Así, por ejemplo, surgieron sistemas capaces de generar ejercicios (siguiendo las ideas de Uhr, 1969) o de adaptar su nivel de dificultad dependiendo del avance del estudiante (Tennyson et al., 1984).

La progresiva mejora de los sistemas CAI, añadiendo más y más técnicas de inteligencia artificial, llevó a acuñar el término *Intelligent Computer-Aided*

Instruction (ICAI) que después fue reemplazado por el hoy aceptado ITS ya indicado (Sleeman y Brown, 1982).

La inteligencia añadida a los CAI consistió en añadir al sistema más y más conocimiento con el que razonar para mejorar la forma de presentar los contenidos, proporcionar pistas, guía y comentarios a los estudiantes. En concreto, los ITSs beben generalmente de tres fuentes de conocimiento: el conocimiento del dominio, el modelo pedagógico y el modelo del estudiante.

Estos tres tipos de conocimiento son los responsables de decidir qué información dar al usuario, y son tan comunes que suelen aparecer en todos los ITSs como módulos software perfectamente delimitados. Existe un cuarto módulo que, si bien no tiene ningún tipo de conocimiento asociado, también aparece en los ITSs, que es el módulo de comunicación, responsable de la presentación de los contenidos al usuario.

Teniendo en cuenta la frecuencia e importancia de los tres tipos de conocimiento y del módulo de comunicación, en los siguientes apartados describiremos su misión y responsabilidades. Posteriormente, veremos cómo las distintas arquitecturas software prototípicas los relacionan.

3.2.1. Módulo experto

En los primeros sistemas CAI, el conocimiento a transmitir estaba *cableado*, almacenado en bloques que Wenger (1987) llama *frames*, y que hoy por hoy suelen llamarse *learning objects*. La aplicación presenta cada uno de esos bloques diseñados por el experto humano cuando se dan las circunstancias adecuadas.

Históricamente, en la transición que tuvo lugar hacia los ITS, el primer paso fue la representación *explícita* de ese conocimiento, en lo que se conoce como *módulo experto*[2]. Su función principal es la de servir de fuente del conocimiento que hay que presentar al estudiante, generar explicaciones, responder a sus preguntas, etc. Una segunda utilidad es ser capaz de *evaluar* la solución aportada por el alumno automáticamente, sin la necesidad de un tutor humano.

Existen distintas alternativas para la representación de este tipo de conocimiento (Dhar y Pople, 1987). Dependiendo de la complejidad con la que se haga, el módulo será capaz de evaluar esas soluciones mejor o peor. Por ejemplo, se puede guardar una base de ejercicios acompañados de su solución (como en Stottler y Vinkavich, 2000) y comparar la solución del usuario con la suya. Otra alternativa es construir un modelo que contenga la estructura y el comportamiento del dominio dinámico (Davis, 1984). En ese caso, se puede hacer una evaluación compleja de la solución, comparando las discrepancias entre el comportamiento predicho por el modelo y el de la solución

[2]Hay que destacar que, a pesar de la similitud de nomenclatura, el módulo no tiene por qué ser un *sistema experto*.

aportada, como hace por ejemplo Kumar (2002).

3.2.2. Modelo del estudiante

En cualquier proceso comunicativo, el orador debe ser consciente del nivel de conocimiento de la audiencia. Los sistemas de enseñanza inteligentes, no escapan a esto, y mejoran si también lo tienen en cuenta.

Para eso, igual que se modela el conocimiento a enseñar, los ITSs suelen incluir un modelo del usuario o estudiante. En realidad, la idea no se restringe a sistemas de enseñanza, sino que se ha utilizado en gran variedad de sistemas, como en los enfocados al filtrado de información (Maes, 1994), comercio electrónico (Abbattista et al., 2002) o interfaces adaptables (Brusilovsky, 2001).

La recolección de los datos utilizados para componer ese modelo puede ser explícita o implícita. El método explícito consiste en preguntar directamente al usuario sus preferencias y de ahí extraer el modelo, mientras que con el método implícito se monitorizan sus acciones mientras interactúa con el sistema. Esa colección de información después será utilizada para adaptar la funcionalidad del sistema, y hacerla más amigable (Kobsa, 1995; Papatheodorou, 2001).

Según Wenger (1987), la idea de modelo de usuario en sistemas de enseñanza se remonta a 1975 (Fletcher, 1975). En estas aplicaciones, el modelo del estudiante debe ser capaz de inferir, a partir de aspectos observables del comportamiento del estudiante, una *interpretación* de sus acciones y ser capaz de reconstruir el *conocimiento* que ha "generado" esas acciones, lo que Hollnagel (1993) llama la extracción del *genotipo* a partir del *fenotipo*.

En los sistemas CAI simples, el modelo del usuario puede ser una mera medida del rendimiento general del estudiante, representada incluso con un único valor real. Sin embargo, en sistemas más complejos o en ITSs, la medida debe disgregarse, para cubrir por separado el nivel de conocimiento de cada uno de los elementos (o conceptos) que el sistema enseña. A este respecto, puede distinguirse, por ejemplo, el modelado del conocimiento basado en los conceptos (*concept-based knowledge modeling*) y el basado en temas (*topic-based knowledge modeling*).

En cualquier caso, los tipos de modelado anteriores almacenan por cada estudiante una representación de su conocimiento. Por el contrario, algunos sistemas (Kay, 1995; Brusilovsky, 1994), no sólo guardan el perfil inferido del estudiante (el "genotipo"), sino que también dejan registrados los eventos o acciones del usuario ("fenotipo") que llevaron a inferir ese modelo. De esta forma, no pierden información, y es fácil añadir reglas nuevas al sistema de inferencia para conocer más acerca de un usuario cuyo comportamiento ha sido registrado.

Para finalizar con el modelo del usuario, resulta de especial relevancia

en nuestro contexto, la reciente aparición de trabajos de investigación en los que se intenta modelar al jugador de un videojuego, para adaptar su comportamiento e intentar mantener alto el nivel de interés que éste tiene (Spronck, 2005; Yannakakis y Maragoudakis, 2005; Houlette, 2004; Charles y Black, 2004).

3.2.3. Módulo pedagógico

Ya se ha mencionado que en cualquier comunicación de contenidos se toman decisiones pedagógicas sobre la secuenciación de dichos contenidos. Esas decisiones pueden ser estáticas o dinámicas, es decir, tomadas previamente o durante la ejecución, teniendo en cuenta las acciones del estudiante.

El dinamismo en estas decisiones es lo que marca la diferencia entre un sistema educativo y un libro. En un libro, la secuencia de contenidos es una decisión del autor que plasma estáticamente, y que no cambia. En una aplicación informática, es el propio sistema el que decide, en base a unas determinadas entradas, qué contenidos presentar al estudiante, de forma que dos usuarios distintos verán una secuenciación de contenidos distinta.

En los sistemas CAI, estas decisiones, a pesar de ser dinámicas, es decir, variar en ejecución según el alumno, estaban *cableadas* en el código del sistema o incluso no existían.

En el área de los ITSs, la selección de esos contenidos es responsabilidad del módulo pedagógico (o de tutoría). Teniendo en cuenta el conocimiento del dominio y el modelo del estudiante, utiliza una serie de reglas para decidir qué debe hacer el estudiante a continuación. Idealmente, las reglas pedagógicas utilizadas deben ser generales y reutilizables independientemente del conocimiento enseñado en el sistema. Si es así, el módulo almacena un conjunto de principios generales expresados explícitamente e interpretados en el contexto específico del sistema para tomar las decisiones concretas.

A nivel global, las decisiones, como hemos dicho, afectan a la secuenciación de contenidos, la selección de los ejercicios a presentar, etc. Sin embargo, si el seguimiento del estudiante es más estrecho, el módulo también debe estar pendiente de las acciones que realiza el estudiante en cada momento. Con este modo de actuar, se puede conseguir que el sistema dé explicaciones contextualizadas o vaya guiando al estudiante en su aprendizaje. Esto último está muy relacionado con los agentes pedagógicos de las secciones 3.2.5 y 3.5.

Un área que ha despertado mucho interés, relacionada con el módulo pedagógico es dotar al sistema de *adaptabilidad* en el entorno de aprendizaje. El módulo pedagógico, que es el que hace las veces de "tutor" cambia el entorno dependiendo del modelo del estudiante, igual que se hace en otras áreas no relacionadas con la enseñanza (ver por ejemplo Gómez Gauchía et al., 2006; Göker, 2003; Göker y Thompson, 2000).

Para terminar, hay que hacer notar que la creación de un módulo de este

tipo no es una tarea fácil. La pedagogía es un arte que requiere una gran versatilidad, y no existe una especificación de sus componentes esenciales, lo que ha contribuido a retrasar la inclusión real de decisiones pedagógicas en los ITSs. Podemos decir que el conocimiento pedagógico ha sido el área olvidada de los sistemas CAI e ITS (Heffernan, 2001).

3.2.4. Módulo de comunicación

Los módulos anteriores son los responsables, en mayor o menor medida, de la selección de los contenidos a presentar al usuario. El módulo de comunicación se preocupa del formato final que se utilizará para mostrarlo al estudiante. Se puede decir que traduce la representación interna del conocimiento al "lenguaje" interpretable por el alumno. También realiza el paso inverso, sirviendo de comunicación entre el estudiante y el sistema, cuando éste recibe las interacciones de aquél.

Las investigaciones en ITSs se han centrado tradicionalmente en el resto de módulos, más cercanos a las teorías educativas y a problemas tradicionalmente difíciles como modelado de usuario o representación del conocimiento. Sin embargo, el módulo de comunicación no debería olvidarse.

Desde el punto de vista del usuario, el interfaz *es el sistema*. Si la forma en la que se presenta un tema lo hace difícil de interpretar, el estudiante puede sentirse sin fuerzas para terminar de leerlo. En definitiva, de su atractivo depende la aceptación que el sistema *completo* tendrá entre los usuarios.

Siendo puristas, podríamos clasificar los tipos de representación en tres tipos: los entornos unidimensionales que únicamente presentan texto, los entornos bidimensionales que complementan el texto con figuras, y los entornos tridimensionales que utilizan un entorno virtual en tres dimensiones para la representación de los contenidos.

Con la progresiva mejora de los equipos, los entornos unidimensionales son ya bastante escasos, mientras que los tridimensionales abundan por doquier. Eso hace que la complejidad del módulo de comunicación se haya visto incrementada hasta el punto de que hoy por hoy el entorno virtual y sus habitantes no pueden ser considerados como un simple módulo de comunicación, sino que debe ser tratado de manera especial dentro de la arquitectura del sistema completo (de Antonio et al., 2004).

El auge de los entornos tridimensionales excede el área de los sistemas educativos. De hecho, existe toda un área de investigación en lo que se conoce como *interfaces virtuales inteligentes* o en inglés *intelligent virtual environments* o IVEs (Aylett y Cavazza, 2001; Aylett y Luck, 2000). Según la definición dada por Aylett y Cavazza, estos son entornos tridimensionales en tiempo real e interactivos que añaden algún tipo de vida o inteligencia artificial. Una de las áreas de investigación en este campo, y que está relacionada con lo que decíamos en la sección anterior, consiste en la personalización que

adecúa la presentación del material al usuario; por ejemplo Chittaro y Ranon (2002) generan distintos ficheros VRML para la generación del entorno virtual dependiendo del perfil del usuario. Aunque esto tiene una utilidad directa en sistemas de enseñanza, también se utiliza en otros contextos. Es el caso de Panayiotopoulos et al. (1998), un entorno virtual inteligente donde se guía por una universidad virtual. En realidad, si el IVE está bien diseñado es posible utilizarlo con cualquier fin, como el sistema AdapTIVE, que puede utilizarse tanto para sistemas educativos (dos Santos y Osório, 2004c) como para comercio electrónico (dos Santos y Osório, 2004a).

Una mejora adicional para los programas educativos que presentan entornos virtuales en los que el usuario *habita* es añadir a ellos una especie de tutor que monitoriza las acciones del estudiante y reacciona ante ellas en el momento adecuado. Son los agentes pedagógicos que describimos en la sección siguiente.

3.2.5. Agentes pedagógicos

Los ITSs son sistemas educativos a los que se ha añadido cierta inteligencia artificial para ayudar en el proceso enseñanza–aprendizaje. En muchos casos, utilizan modelado del usuario y son capaces de adaptar las estrategias dependiendo de la situación.

Aunque los ITSs están pensados para simular el comportamiento de un profesor, la realidad es que tradicionalmente fallaban en el modo de interacción, pues ofrecían un estilo rígido y fuertemente dirigido por la aplicación.

Para mejorar esta situación, se empezaron a añadir *agentes inteligentes*, que en el contexto en el que estamos, son conocidos como *agentes pedagógicos*. Estos agentes son entidades cuyo propósito fundamental es mejorar la interacción y comunicación entre el sistema y el estudiante, jugando el papel de *tutor* o profesor (Johnson et al., 2000). Pueden además adquirir otras responsabilidades, como monitorizar al estudiante y ayudarle durante el proceso de aprendizaje. Es más, también suelen intercambiar información con ellos, pueden adaptar la presentación de los conceptos dependiendo de la situación y alumno actual, etc.

El concepto de agente pedagógico suele venir acompañado de un interfaz visual que dota al agente de una fuerte apariencia humana, o al menos la de un ser vivo e inteligente. Son lo que se conocen como *agentes pedagógicos animados*, para distinguirlos de aquellos pocos agentes pedagógicos sin presencia visual.

Añadir un agente pedagógico animado al entorno aumenta los tipos de interacción hombre–máquina que pueden realizarse y que no se tienen en sistemas educativos normales. Los agentes deben ser capaces de interactuar con el estudiante de manera creíble. Dependiendo de la complejidad del agente pedagógico, éste soportará más o menos tipos de interacciones, como por

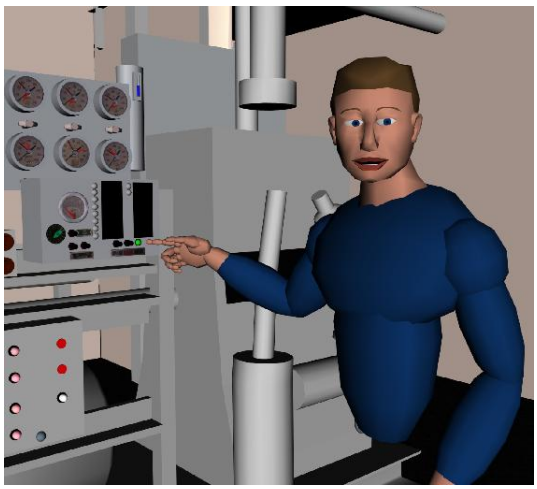


Figura 3.1: Captura del sistema Steve en funcionamiento

ejemplo, hacer demostraciones interactivas, guiar dentro del entorno virtual, usar expresiones para guiar la atención durante las conversaciones, o usar expresiones corporales para guiar la atención (Rickel y Johnson, 2000; Rickel, 2001). Analizaremos con un poco más de detalle cada una de ellas en los siguientes puntos.

3.2.5.1. Demostraciones interactivas

En general, demostrar cómo se realiza una tarea es mucho más efectivo que intentar describir cómo hacerla y que el estudiante memorice cada uno de los pasos. Eso es especialmente cierto cuando la tarea involucra habilidades motoras; incluso únicamente con ver cómo se realiza hace que sea más probable que el alumno lo recuerde en un futuro.

En un entorno de aprendizaje donde el agente comparte su espacio vital con el estudiante, las demostraciones por parte del agente son incluso más beneficiosas. La demostración pasa a ser interactiva, al contrario que lo que ocurre cuando se muestra un simple vídeo, ya que el usuario puede seguir moviéndose libremente por el entorno, ver la explicación desde distintas perspectivas, interrumpir la exposición para realizar preguntas, o incluso solicitar terminar por él mismo la tarea.

Un ejemplo de agente con demostraciones interactivas es Steve (Rickel y Johnson, 1999, 2000), que habita en una simulación virtual de un navío estadounidense donde los estudiantes aprenden procedimientos de actuación. En la figura 3.1 se puede ver una captura en la que aparece el agente pedagógico explicando una determinada acción. Steve utiliza planificación para adaptar sus demostraciones dependiendo del estado del entorno, proporcio-

nando unas demostraciones mucho más adecuadas que vídeos de actuación tradicionales.

3.2.5.2. Guía de navegación

Dejar al usuario en un entorno virtual tiene el peligro de que se sienta perdido, sobre todo si el entorno es demasiado grande o contiene numerosos objetos con los que interactuar.

El agente pedagógico puede servir como guía de navegación por estos mundos virtuales, al ir mostrando las tareas a realizar y moverse de un sitio a otro para alcanzar las piezas con las que operar.

Otro aspecto importante a tener en cuenta es que si el entorno de aprendizaje está basado en un espacio real, como puede ser la sala de máquinas de un barco (Rickel y Johnson, 1998), el estudiante estará aprendiendo dónde están situados los componentes principales sin necesidad de haber estado *in situ* en ese lugar. El alumno puede construirse un modelo espacial de la zona de trabajo que es imposible conseguir en una clase tradicional.

Otro ejemplo en el que una de las funciones del agente es la de hacer de guía por el entorno virtual es WHIZLOW, un agente pedagógico desarrollado por Lester et al. (1999b), que habita la "CPU CITY", una ciudad virtual tridimensional que representa el interior de un ordenador. El agente, entre otras cosas, guía al estudiante por las distintas zonas de la ciudad.

3.2.5.3. Expresiones para guiar la atención

Cuando una persona está hablando utiliza numerosas expresiones para guiar la atención del oyente, como mover los brazos, o hacer gestos con la cara.

Del mismo modo, los agentes pedagógicos pueden utilizar ciertas expresiones corporales para guiar la atención del estudiante. En este sentido, pueden señalar a los objetos cuando están explicando algo sobre ellos, mirar un objeto mientras lo manipulan, o mover la cabeza para dar la sensación de que miran hacia un piloto indicador cuando quieren comprobar si está encendido o no. Por ejemplo, Nijholt y Heylen (2002) implementan un agente que mira hacia una mesa mientras apunta que en ella se puede encontrar folletos con la información solicitada.

3.2.5.4. Comunicación no verbal

Todos tenemos la imagen de un niño temeroso, antes de hacer algo, mirando con cara interrogante al padre o profesor, para ver si capta alguna expresión del adulto acerca de lo acertado o desacertado de la acción que pretende realizar a continuación.

En realidad, uno de los papeles principales de cualquier tutor es proporcionar *feedback* ante las acciones del estudiante, ya sea usando comunicación verbal o no verbal. De esta forma, el tutor influencia al estudiante, o le hace saber si la acción que está apunto de realizar es adecuada o no.

Los gestos pueden ser simples movimientos de cabeza, afirmando o negando, aunque también pueden ser muecas de agrado o desagrado. Más aún, las expresiones pueden utilizar todo el cuerpo. Por ejemplo se pueden dar saltos para mostrar alegría, o mostrar peligro alejándose del objeto que el usuario va a tocar.

Este tipo de comunicación no verbal es muchas veces más adecuada que un comentario hablado pues es menos intrusiva, pero consigue el mismo efecto: que el estudiante se lo piense dos veces antes de dar el siguiente paso.

Un ejemplo de esto es Adele, desarrollado por Shaw et al. (1999), que sonríe o hace gestos afirmativos cuando el usuario realiza las acciones correctamente y muestra desagrado cuando se equivoca. También HERMAN THE BUG (Lester et al., 1999a) muestra su contento haciendo movimientos de alegría por la pantalla. Ambos son ejemplos de agentes pedagógicos en dos dimensiones.

3.2.5.5. Gestos durante las conversaciones

Es una parte de la comunicación no verbal tiene la suficiente importancia como para tratarla aparte. Cuando hablamos, empleamos inconscientemente una amplia variedad de gestos acompañando a la voz (Wachsmuth y Kopp, 2002). Algunos de ellos están estrechamente ligados a ella, como mover las cejas o la cabeza al elevar la entonación, mientras que otros ayudan a regular la conversación, como el uso del contacto visual, cuando en una pausa el hablante pierde el contacto y mira al suelo, o lo recupera para pedir la respuesta.

Un ejemplo de agente con estas características es el creado por Cassell et al. (2001), que utiliza la mirada y los gestos para controlar el turno en la conversación, mueve la cabeza para hacer énfasis en ciertas palabras, y alguna característica parecida.

Aparte de estos cinco tipos de interacción, existen otras capacidades interesantes descritas por Rickel (2001). Por ejemplo, es importante la existencia de avatares e historias interesantes para atrapar la atención del estudiante, como se describe en Gómez Martín et al. (2004b). Rickel también destaca que los agentes pueden hacer de compañeros virtuales. No obstante, pensamos que esta capacidad está más cercana a otro tipo de agentes, que hacen de *coaprendices*, es decir, avatares que "fingen" aprender al mismo tiempo que el estudiante (Goodman et al., 1998).

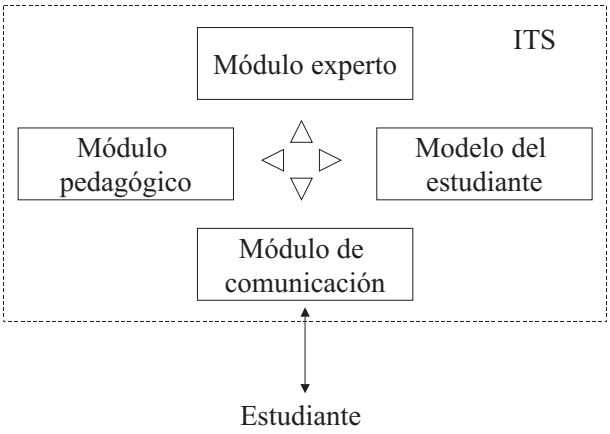


Figura 3.2: Arquitectura clásica de un ITS (Wenger, 1987)

3.3. Arquitecturas de ITSs

En el apartado anterior veíamos los tres tipos de conocimiento más relevantes para un ITS y las atribuciones dadas al responsable de la comunicación con el usuario. En este apartado, pasamos a describir tres tipos de arquitecturas típicas en los ITSs.

3.3.1. Arquitectura clásica de ITSs

Una vez presentados los componentes básicos clásicos que se encuentran en la mayoría de los ITSs, es fácil describir su arquitectura clásica presentada por Wenger (1987) (ver figura 3.2).

En ella, se divide el sistema, precisamente, en los cuatro módulos vistos previamente:

- Módulo experto: almacena el conocimiento y posee la habilidad de solucionar los problemas del dominio.
- Modelo del estudiante: diagnostica la habilidad del estudiante en el dominio enseñado.
- Módulo pedagógico: responsable de las estrategias pedagógicas a seguir.
- Módulo de comunicación: implementa el interfaz hombre–máquina.

Un ejemplo de escenario de interacción entre los módulos puede ser el siguiente: el módulo pedagógico elige la siguiente tarea a realizar basándose en la información proporcionada por el módulo del estudiante; el módulo de

comunicación se lo muestra y traduce la entrada de éste para que pueda ser procesada por el módulo experto, el cual intenta diagnosticar algún posible problema del alumno al resolver el problema en cuestión. Este diagnóstico se utilizará para actualizar el modelo del estudiante, y repetir el ciclo.

3.3.2. Arquitectura Modelo de conocimiento–Vista

Devedzic y Harrer (2005) en su esfuerzo por extraer patrones software en las arquitecturas de ITSs, identificaron lo que vinieron a llamar la arquitectura "modelo de conocimiento–vista" (en inglés *knowledgemodel–view*).

Esta arquitectura se basa en la idea de que existen dos tipos distintos de módulos, los responsables principalmente de albergar conocimiento, y los *consumidores* de esos datos:

- Modelizaciones del conocimiento: que se corresponden con los tres primeros módulos de la arquitectura clásica: el conocimiento sobre el dominio, el modelo del estudiante y el modelo pedagógico (o reglas pedagógicas).
- Componentes para la representación de la información al usuario y para interactuar con él. En la terminología habitual, estos componentes se conocen como *vistas* (Gamma et al., 1995).

La realidad es que esta arquitectura está muy relacionada con la variante documento–vista (ver Buschmann et al., 1996, página 140) del patrón más general modelo–vista–controlador (Buschmann et al., 1996). La única diferencia es que en estos patrones clásicos, únicamente existe un modelo, mientras que en un ITS se pueden tener distintos modelos simultáneamente (dominio, estudiante y pedagógico).

El patrón general considera que los tres tipos de conocimiento pueden formar su propio modelo, de forma que cualquier otra parte del sistema puede jugar el papel de "vista" de ese modelo, y ser informado cuando ocurre un cambio.

Un ejemplo de un sistema que sigue esta arquitectura es la arquitectura centrada en el modelo del estudiante de Brusilovsky (1994, 1995). En ella, no obstante, existe un único modelo, el que almacena el conocimiento que se tiene del usuario. Sobre él gira el resto de componentes, que son las "vistas" asociadas. Cuando alguna de ellas altera el modelo, esos cambios son redistribuidos al resto. La arquitectura establece dos tipos de vistas, aquellas que guardan el resto de tipos de conocimiento (y que Brusilovsky llama *agentes*) y el resto de módulos o "herramientas de interfaz", que permiten la interacción con el usuario.

Igual que la anterior, otra variación de la misma idea consiste en centrar todo el sistema en torno a la *simulación* del entorno virtual en el que se

desarrolla el aprendizaje (Munro et al., 1997, 1999). En este caso, las vistas son las distintas posibles representaciones del entorno.

Como veremos en el capítulo 6, nuestro sistema JV^{2}M utiliza también la misma idea de modelo y vista.

3.3.3. Arquitectura basada en agentes

La arquitectura clásica divide el problema de crear un sistema de tutoría inteligente en cuatro módulos claramente diferenciados con unas tareas asignadas perfectamente delimitadas.

Sin embargo, el desarrollo de estos sistemas es cada vez más complejo, especialmente cuando se le añaden entornos virtuales, por lo que se necesitan otras soluciones tomadas de la Ingeniería del Software.

Una de las soluciones es recurrir a la aproximación basada en agentes (Giraffa y Viccari, 1998). En realidad los sistemas que siguen esta alternativa suelen ser variaciones de la arquitectura clásica, pero donde cada uno de los módulos es convertido en uno o varios agentes especializados que implementan la misma funcionalidad. En ese caso, los agentes se comunican entre sí para conseguir la funcionalidad completa usando mecanismos *ad hoc* o plataformas de agentes como JADE (Bellifemine et al., 2001a,b), creando lo que Giraffa y Viccari llaman *sociedad interna* del sistema. A pesar de la existencia de esta "sociedad", desde el punto de vista del usuario el sistema es una aplicación indivisible, igual que lo era con la arquitectura clásica.

El uso de tecnología de agentes en las arquitecturas de ITSs va de la mano del uso de agentes pedagógicos animados. La mayoría de los sistemas que incluyen este tipo de agentes dicen estar fundamentados en una arquitectura basada en agentes. Por eso, existe el peligro de confundir el significado de la palabra "agente", ya que en unos casos se referirá a alguno de los componentes del sistema, mientras que en otros se referirá al personaje virtual que hace de tutor y todo lo que lleva detrás. En el caso de estos dos tipos de agentes, se habla de que en el sistema existen dos sociedades distintas, la *sociedad interna* comentada antes, y la *sociedad externa* formada entre los agentes virtuales cuando se comunican a través del entorno virtual; a diferencia de la primera, esta última sí es apreciable por el usuario.

Como ejemplo de este tipo de arquitecturas podemos nombrar a Reyes y Sison (2000) donde todos los componentes son agentes, y a de Antonio et al. (2005). En ésta última, cada uno de los módulos de la arquitectura clásica de la sección 3.3.1 pasa a ser un agente; además, permite múltiples estudiantes trabajando juntos, cada uno con su agente de comunicación (haciendo las veces de módulo de comunicación). Todos estos agentes se conectan con un agente central que controla la comunicación completa, y que es el único que se comunica con los agentes responsables de la tutoría, modelo de los estudiantes y conocimiento del dominio.

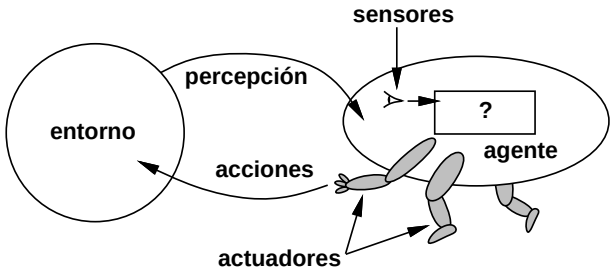


Figura 3.3: Estructura de un agente simple (Russell y Norvig, 1995)

3.4. Arquitectura de entornos virtuales con agentes

La construcción de un agente pedagógico animado inmerso en un entorno virtual tiene las dificultades típicas de la construcción de agentes, de las aplicaciones de realidad virtual y de los sistemas de tutoría inteligentes (ITSs). La construcción de ITSs ya ha sido tratada en la sección anterior, y la construcción de entornos virtuales ha sido tratada en el capítulo anterior, dentro de las arquitecturas de videojuegos.

En esta sección mostramos ejemplos de arquitecturas diseñadas para añadir a los entornos virtuales agentes en general, mientras que en la sección siguiente abordaremos los agentes pedagógicos.

En realidad, los agentes en entornos virtuales recuerdan mucho a los caracteres no jugador (NPCs) controlados por la inteligencia artificial de un videojuego. Sin embargo, para su implementación no se utiliza tecnología de agentes, por lo que desde el punto de vista del desarrollador son distintos.

Cuando se habla de agentes en general, se piensa en entidades software que captan el entorno por medio de sensores y que actúan sobre él a través de una serie de actuadores (Russell y Norvig, 1995; Wooldridge, 1999). Un esquema simplista aparece en la figura 3.3.

Un aspecto importante es la *autonomía*, que marca la diferencia entre los programas tradicionales y los agentes (Franklin y Graesser, 1996). Según Franklin y Graesser, un agente autónomo es un sistema situado dentro de un entorno del que forma parte. A lo largo del tiempo, el agente va percibiendo y actuando sobre él. Sus acciones están dirigidas hacia la consecución de una lista de objetivos. Las acciones son elegidas de tal forma que las situaciones futuras a las que se llega con ellas están más cerca del objetivo final.

En la figura 3.3, el signo de interrogación simboliza el módulo estratégico del agente que decide qué acciones realizar en el entorno dependiendo de lo percibido. La función del diseñador de la inteligencia artificial consiste, precisamente, en diseñar esa función que mapea las percepciones en acciones. Los agentes más simples tendrán una simple batería de reglas condición–acción;

los agentes más sofisticados mantendrán un estado interno, y decidirán sus acciones basándose en objetivos, aplicando búsqueda y planificación.

Cuando el agente tiene una representación virtual en forma de avatar dentro de un entorno tridimensional, las acciones tienen que verse reflejadas en ese personaje. Esto puede llegar al extremo de Thalmann (2001), donde la comunicación entre los agentes (avatares virtuales) *no* se realiza utilizando los métodos típicos de comunicación entre agentes, sino que un agente se comunica con otro cambiando la forma de actuar de su avatar. El segundo avatar percibirá en el entorno ese comportamiento, y lo interpretará como un mensaje.

Uno de los retos más importantes para estos sistemas es la definición del interfaz entre la parte del agente que toma las decisiones y la parte responsable de animar su personaje. En el siguiente apartado trataremos algunos de los aspectos generales, para posteriormente describir la arquitectura de algunos sistemas que utilizan entornos virtuales con agentes animados.

3.4.1. Interfaz entre el agente y el entorno

Como ya hemos dicho, un agente inteligente observa el entorno, guarda una representación interna del mismo, y decide qué operaciones ejecutar sobre él para realizar las tareas para las que ha sido diseñado.

En los agentes en general, el número de actuadores que el módulo estratégico tiene disponibles limita las acciones que puede realizar sobre el entorno en el que se encuentra. Cuando el agente está inmerso, o mejor dicho, controla un avatar que aparece inmerso en un entorno virtual, otra de las tareas del agente es el control del propio personaje. A la hora de pensar en la arquitectura, debe dejarse claro el conjunto de *primitivas* que el agente tiene disponibles para controlarlo. Dependiendo de su número y complejidad, el agente podrá ser más o menos completo. Por ejemplo, si el avatar no es capaz de cambiar el gesto de la cara (para sonreir, o mover los ojos para mirar a otro sitio), no tiene sentido añadir emociones o control de la mirada al agente.

Algo parecido ocurre con los sensores. Dependiendo de la cantidad de cosas que el agente pueda percibir del mundo, su comportamiento podrá adecuarse convenientemente a más o menos estados del mundo. Cuando se diseña el sistema, se debe tener claro a qué tipo de situaciones se desea que el agente pueda reaccionar para que el entorno virtual pueda comunicárselo a través de sus sensores.

Cuando estudiemos los agentes pedagógicos en la sección 3.5, listaremos los tipos de percepción más importantes que debería tener uno de estos agentes. Por el momento, nos limitaremos a decir que un agente inmerso en un entorno virtual necesita, al menos, conocer el entorno por el que se está moviendo para poder ir de un sitio a otro sin colisionar con nada, y conocer

dónde están los objetos con los que puede interaccionar, es decir, en qué lugares están los actuadores. Para el primer cometido hay tres aproximaciones, que explicamos por orden creciente de complejidad:

- El agente no necesita saber cómo está organizado el entorno virtual. Cuando quiere realizar alguna acción sobre el mundo, se lo comunica al motor responsable de la apariencia del mundo virtual, y él se encarga de mover al avatar y hacerle interactuar con los objetos adecuados. Tiene la ventaja de que un cambio en el aspecto del mundo virtual no requiere una modificación en el agente. Si hacemos la analogía agente virtual – personaje no jugador en un videojuego, este modo de implementar el movimiento es utilizado en mucho de ellos (por ejemplo, el Unreal Tournament de Epic Games (1999)), en donde el *comportamiento* se define en base a primitivas de movimiento de alto nivel, resueltas por el motor de juego subyacente.

- El agente "nace" conociendo dónde están los objetos con los que interactúa. El método más sencillo es representar este conocimiento en forma de grafo de adyacencia. Cada nodo del grafo representa un punto (localización) en el entorno virtual, y cada arista indica que se puede andar entre los dos nodos sin peligro de colisionar. Para ir de un lugar a otro se utiliza el algoritmo de Dijkstra (Dijkstra, 1959), o cualquier otro algoritmo de búsqueda. Este método funciona únicamente para mundos estáticos, o con cambios localizados que no hagan aparecer objetos en medio de las rutas entre localizaciones, que hagan variar el grafo. Lo han utilizado, por ejemplo, dos Santos y Osório (2004b) en su entorno *AdapTIVE*.

- El agente no conoce la geografía del mundo. Técnicamente, es posible extraer el plano o mapa de un laberinto utilizando las imágenes generadas por el motor gráfico. Utilizando métodos de visión sintética, Monsieurs et al. (1999) analizan las imágenes renderizadas desde el punto de vista del agente, para averiguar la distribución de las paredes. En realidad, en ese sistema, la inteligencia de la aplicación se centra en ser capaces de averiguar ese mapa para poder navegar por un entorno completamente desconocido. Si bien tiene una aplicación práctica en otras áreas (robótica), en agentes pedagógicos no tiene sentido utilizar estas técnicas, pues el avatar tendría primero que aprender la estructura del mundo; al fin y al cabo, no tiene sentido pensar que un tutor humano tenga primero que aprender dónde están las cosas, antes de poder enseñar como utilizarlas.

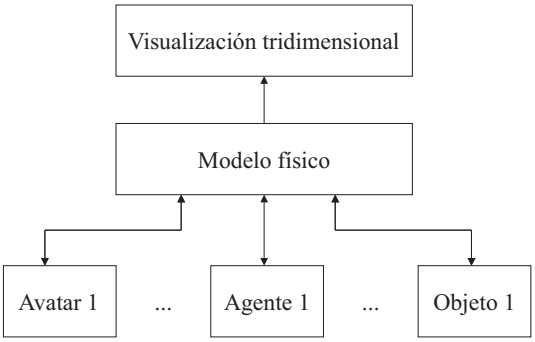


Figura 3.4: Arquitectura de SimHuman (Vosinakis y Panayiotopoulos, 2001b)

3.4.2. Ejemplos de agentes en entornos virtuales

Durante los últimos años, existe mucho movimiento en el área de agentes virtuales inteligentes (o *Intelligent Virtual Agents*, *IVA*). En muchos casos, el interés se centra en dar una apariencia más humana a los avatares controlados por tales agentes. A este respecto, existen muchos trabajos sobre emociones, mejora de animaciones de avatares, del cálculo de rutas para hacerlas más humanas, mantenimiento del contacto visual entre el agente y el usuario, etc.

Nuestro interés se centra sin embargo en la arquitectura software del sistema completo que permite a los agentes interactuar con el entorno virtual y que establece el tipo de cosas que percibirán y cómo podrán actuar sobre el mundo que habitan.

En el capítulo anterior veíamos que en el mundo de los videojuegos no se ha venido haciendo tan explícita la división entre el agente o entidad software que define el comportamiento de un personaje no jugador y la representación tridimensional del mismo.. En este apartado veremos dos ejemplos de entornos virtuales que incorporan agentes inteligentes, para ver cómo se hace esta división fuera del contexto de los videojuegos.

3.4.2.1. SimHuman

SimHuman (Vosinakis y Panayiotopoulos, 2001b) es una plataforma para la generación de entornos tridimensionales en tiempo real con agentes virtuales. Permite definir escenas en tres dimensiones con agentes inteligentes y avatares controlados por el usuario. De esta forma, sirve como base a aplicaciones de entornos virtuales y sistemas de simulación.

Para representar el mundo, tanto los objetos estáticos como los avatares están almacenados en forma de colección de polígonos que son dibujados utilizando un módulo específico de visualización. Para los personajes, además, se tiene una biblioteca de animaciones, así como su esqueleto para poder uti-

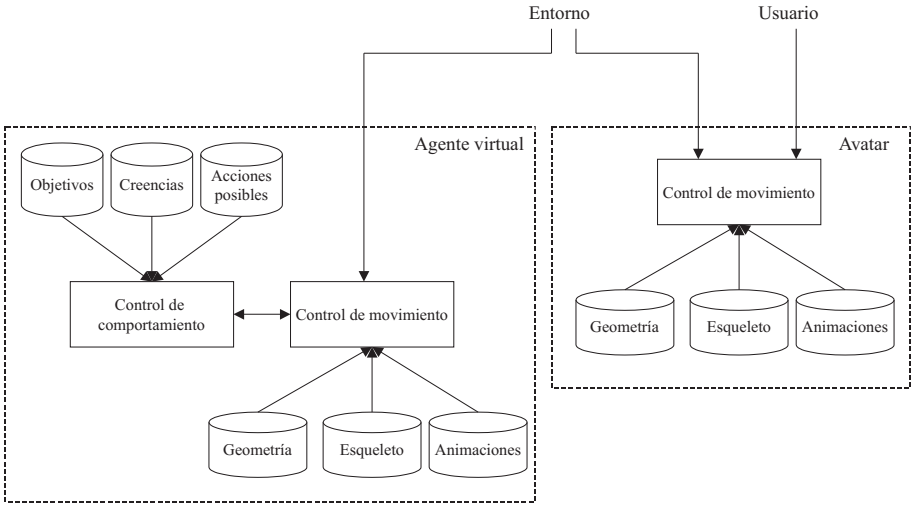


Figura 3.5: Estructura interna de un avatar en SimHuman (Vosinakis y Panayiotopoulos, 2001b)

lizar cinemática inversa y ejecutar animaciones dinámicas que permiten, por ejemplo, coger objetos en movimiento (Vosinakis y Panayiotopoulos, 2001a).

Esta representación visual se ve complementada con un motor físico que calcula en cada fotograma la nueva posición de cada objeto, de acuerdo con su masa, posición y velocidad actual. El módulo además, posee un sistema de detección de colisiones que detecta si dos objetos entran en contacto para alterar sus velocidades.

En la arquitectura de SimHuman (ver figura 3.4), el tercer y último componente es el control de cada uno de los objetos, tanto los estáticos como los dinámicos. En el caso de los avatares, el control se preocupa de cambiar la animación dependiendo del momento.

Con esto, se consigue un mundo virtual poblado con una serie de avatares con apariencia humana, movimientos coherentes y que puede ejecutar una serie de acciones en un entorno tridimensional.

Cada uno de los avatares que aparecen puede ser controlado por un usuario humano o por un agente. La estructura interna de cada uno de los dos tipos aparece en la figura 3.5. Independientemente de quién lo controle, cada avatar tiene un módulo encargado de su movimiento (el "*motion controller*"), que tiene tres fuentes de datos: el fichero con la geometría, el esqueleto y la librería de animaciones.

Cuando un usuario controla uno de estos personajes, las órdenes que emite a través del dispositivo de entrada son enviadas al controlador de movimiento que desplaza al avatar de acuerdo a ellas.

Cuando el avatar es manejado por un agente inteligente, el controlador

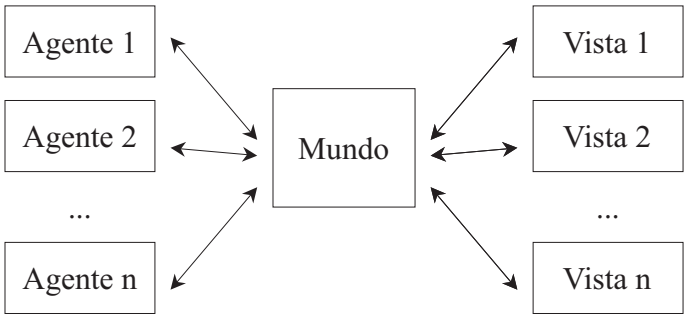


Figura 3.6: Estructura de mVital, según Anastassakis et al. (2001a)

del comportamiento ("*behavioural controller*") es el que emite las órdenes. Este controlador se crea utilizando técnicas de programación de agentes. En particular, el agente almacena una representación simbólica del mundo, y utiliza el ciclo sentir–decidir–actuar para funcionar.

La parte inteligente del agente posee un planificador en Prolog que decide qué acciones ejecutar. Su misión es elaborar los planes o secuencias de acciones que se irán ejecutando en orden. Además, genera el estado del mundo que espera encontrar cuando termine cada una de ellas. De esta forma, si se producen cambios inesperados no producidos por sus propias acciones, puede reaccionar ante ellos.

La arquitectura está pensada para que el ciclo sentir–decidir–actuar se ejecute en paralelo a la simulación del entorno virtual. De esta forma, si el planificador consume mucho tiempo decidiendo qué acción ejecutar cuando se acabe la actual, no bloquea el mundo virtual que está siendo generado.

3.4.2.2. mVITAL

La arquitectura SimHuman, aunque tiene cierto cuidado en definir cómo está implementado cada uno de los agentes inteligentes, está más centrada en la parte de los entornos virtuales inteligentes (IVEs), de forma que permite implementar sistemas habitados por actores con apariencia humana que se mueven de forma creíble en tiempo real.

El mismo grupo de investigación de SimHuman, desarrolló casi simultáneamente mVITAL (Anastassakis et al., 2001a), que permite crear sistemas multi–agente, y añadirlos a un entorno virtual. Al contrario que SimHuman, se centran más en la inteligencia y razonamiento.

mVITAL surge como una mezcla de dos arquitecturas desarrolladas por el mismo equipo, la arquitectura DIVA (Vosinakis et al., 1999), y la arquitectura VITAL (Anastassakis et al., 2001b), esta última preparada para un único agente.

Existen tres componentes básicos que son ejecutados como distintas aplicaciones y conectados mediante sockets siguiendo la filosofía cliente/servidor. Los componentes, que pueden verse en la figura 3.6, son el servidor del mundo, los agentes y las vistas o visores.

Cada uno de los componentes almacena una representación del mundo, que puede ser o bien orientada a objetos o simbólica. El primer tipo de representación trata el entorno como una serie de regiones interconectadas que contienen uno o más objetos. A su vez, cada objeto tiene un nombre, una clase a la que pertenece y una serie de propiedades. Incluso, los mismos agentes se consideran objetos en este tipo de representación.

La representación simbólica utiliza una sintaxis parecida a la de los lenguajes de programación lógicos, como por ejemplo `at(Objeto, X, Y)`, que indica que el `Objeto` está en la posición (`X`, `Y`). Esta representación es la utilizada por los agentes, ya que su motor de inferencia está hecho en Prolog, igual que en SimHuman.

El componente central de la arquitectura es el servidor del mundo, que representa el entorno virtual dentro del que operan los agentes. Se encarga de coordinar el intercambio de datos entre el resto de la aplicación, asegurando que la percepción que tienen los agentes del mundo coincide con la representación visual que aparece en los visores. Para eso, manda la información sensorial a cada uno de los agentes conectados al entorno, y la información visual a cada una de las vistas. Por otro lado, cada agente que ejecuta una operación envía la acción al servidor, para que éste la propague al resto de los elementos.

La vista o visor encapsula la funcionalidad de visualización del sistema. Mantiene un modelo orientado a objetos del entorno, y recibe las notificaciones de cambios en el mundo desde el servidor.

El agente es el componente más importante de la arquitectura, y es el que introduce inteligencia en el sistema, para mover a los actores por el mundo. Al igual que en SimHuman, los agentes utilizan el ciclo sentir–decidir–actuar. En la primera fase, el agente solicita al servidor del mundo una representación pseudo–simbólica del estado, que es procesada y añadida a la base de conocimiento del agente. En la fase de decisión, el motor lógico razona sobre los contenidos de la base de conocimiento y crea un plan para conseguir alguno de los objetivos del agente. Por último, en la fase de actuación, la siguiente acción a realizar se manda al servidor del mundo, también en forma pseudo–simbólica, y se actualiza la base de conocimiento para que refleje la acción que se ha hecho.

Las posibles acciones que el agente puede hacer están prefijadas: sentir (que es enviada al servidor del mundo al principio del ciclo), moverse, coger un objeto, soltarlo y usarlo. Toda la comunicación con el servidor del mundo consiste en una de estas acciones, aunque el significado específico puede variar en cada caso. Así por ejemplo, utilizar una puerta puede ser abrirla o cerrarla,

mientras que utilizar un interruptor puede significar encender la luz. En JV^{2}M, el sistema que describiremos en el capítulo 6 aparecen acciones muy parecidas a estas (sección 6.4).

3.5. Agentes pedagógicos

Como ya se ha dicho, los agentes pedagógicos surgen al unir dos áreas de investigación en principio independientes: los agentes animados que mejoran el interfaz de una aplicación (Hayes-Roth y Doyle, 1998) y el software educativo (Carbonell, 1970; Sleeman y Brown, 1982; Wenger, 1987). La combinación resultante son los agentes pedagógicos animados, definidos por Johnson et al. (2000) como agentes autónomos que dan soporte en el aprendizaje interactuando con los estudiantes en los entornos de enseñanza interactivos.

Los agentes pedagógicos monitorizan el estado entorno, eminentemente dinámico, buscando las oportunidades de enseñanza que puedan surgir. Dependiendo de su complejidad, pueden ser utilizados para aprendizaje tanto individual como cooperativo, soportar interacciones complejas con el estudiante, ayudar en su motivación, etc.

En la sección 3.2.5 veíamos algunas de las ventajas que aportan a los sistemas educativos. En este momento nos planteamos qué tipo de arquitectura software debe implementarse para conseguir toda esa funcionalidad. Para empezar, describiremos qué necesidades especiales tienen los agentes pedagógicos animados que no presentaban los agentes inteligentes vistos en la sección anterior, de forma que entendamos cómo afecta al interfaz que existirá entre él y el entorno. Posteriormente pasaremos a describir la arquitectura típica más utilizada de agentes pedagógicos, llamada por Devedzic y Harrer (2005) el patrón *generalizado de agentes pedagógicos*. Por último describiremos la arquitectura Steve, el agente pedagógico que aparecía en la figura 3.1.

3.5.1. Interfaz entre el agente pedagógico y el entorno

En la sección 3.4.1 describíamos las generalidades del interfaz entre un entorno virtual y un agente inmerso en él. En esta sección nos centramos en los agentes pedagógicos animados, que tienen algunas necesidades especiales que no tienen por qué necesitar los agentes animados de otros contextos.

En los agentes pedagógicos animados, el avatar se encuentra inmerso en el entorno de aprendizaje que comparte con el estudiante. Igual que para los avatares de otras aplicaciones, la capacidad de percepción limita el conocimiento que el agente tiene acerca de sus alrededores, y el conjunto de acciones que es capaz de realizar coartan su comportamiento y capacidad pedagógica. Por ejemplo, en Herman the Bug (Stone y Lester, 1998), el agente es capaz de elegir la música de fondo que sonará para acompañar

a sus acciones lo que, aunque no significa un incremento en su capacidad pedagógica, sí añade en cierto sentido credibilidad y da más o menos nivel de trascendencia a sus acciones, lo que al fin y al cabo define parte de su comportamiento.

Johnson et al. (2000) lista de las percepciones posibles que puede necesitar un agente pedagógico:

- **Estado del entorno**: el agente debe conocer el estado en el que se mueve. Para eso puede preguntárselo a algún módulo responsable de la simulación del mundo, o puede ser informado cada vez que algún atributo de algún objeto cambia.
- **Acciones del estudiante**: cuando está monitorizando, debe percibir cuándo el estudiante realiza alguna acción sobre el entorno. De esta forma no tiene que inferir las acciones que ha realizado utilizando los cambios en el mundo virtual, y puede compararlas con las que habría hecho él en su lugar.
- **Acciones del agente**: igual que con el estudiante, el agente debe percibir cuándo el avatar que le representa en el entorno ha realizado alguna acción, debido a que en muchos casos, el agente manda las acciones a una capa que controla sus movimientos, y tiene que esperar a que su personaje en el mundo termine de realizar la acción para empezar con la siguiente.
- **Preguntas del usuario**: en ciertas ocasiones, el estudiante puede preguntar al agente qué acción debería ejecutar a continuación y por qué.
- **Posición del estudiante y dirección en la que mira**: de esta forma, el agente puede modificar su posición para que el usuario le vea mejor. Además, así puede saber si el usuario está viendo el objeto que va a utilizar seguidamente en su explicación, para en caso contrario esperar a que entre en su campo de visión antes de proseguir el discurso.
- **Eventos de generación de voz**: en agentes sofisticados que poseen generación de discurso a partir de textos, puede ser interesante recibir mensajes del generador para sincronizar la posición de los labios en cada fonema emitido.
- **Frases del estudiante**: en ciertos sistemas, el alumno puede comunicarse con el agente mediante la voz. Un sistema de reconocimiento indicará al tutor virtual la pregunta que el usuario ha hecho. Además, cuando la persona empieza a hablar, el intérprete, aunque obviamente aún no puede decidir qué es lo que el usuario dice, puede mandar al agente un mensaje de aviso, para que éste dirija su atención al usuario, y le mire.

Para alterar el mundo por el que se mueven, los agentes pedagógicos necesitan un conjunto de acciones motoras. Normalmente estas acciones caen dentro de una de estas tres categorías: control del cuerpo del avatar, control del entorno de aprendizaje y habla.

El movimiento del cuerpo del personaje puede consistir simplemente en reproducir secuencias de vídeo grabadas previamente, o descomponerse en acciones primitivas como expresiones faciales, manipulación de objetos y desplazamientos de un sitio a otro.

El control del entorno de aprendizaje son aquellas manipulaciones que el avatar realiza en él. Por ejemplo, cuando el avatar se acerca a un botón y lo pulsa, se debe informar al simulador del mundo que ese botón ha sido presionado, para que se cambie su estado de forma acorde a las reglas que lo rigen; de esta forma, si el botón encendía una luz, el simulador deberá hacer que la luz se ilumine, y mandar el cambio de estado al agente. Las acciones sobre el entorno no tienen por qué estar restringidas a acciones físicas; por ejemplo el cambio de la música de fondo que realiza HERMAN THE BUG (Stone y Lester, 1998) entra dentro de esta categoría.

Por último, el habla es creada típicamente como una cadena, que puede enviarse a un sintetizador de voz para que el estudiante lo escuche, o se le presenta como texto en la pantalla.

3.5.2. Arquitectura de ITSs con agentes pedagógicos

Como agente que habita en un entorno virtual, un agente pedagógico animado debe ser capaz de interactuar con el entorno, por lo que se le presentan los mismos problemas o cuestiones que planteábamos en la sección 3.4 al hablar de agentes en entornos virtuales generales.

Por ello, todos los agentes pedagógicos analizados poseen al menos un componente encargado de la comunicación. Dependiendo del sistema particular, la comunicación estará integrada en un único módulo, o separada en dos, uno que agrupa las acciones que puede realizar el agente sobre el entorno (es decir, los actuadores), y otro que se encarga de la percepción (sensores). El nivel de abstracción de las acciones y percepciones depende, una vez más, del sistema. En algunos casos se puede quedar en acciones de alto nivel como escribir un mensaje, mostrar una pista o actualizar una barra de progreso (Devedzic y Harrer, 2005), mientras que en otros casos consistirá en órdenes de bajo nivel al avatar, como ocurría en SimHuman (Vosinakis y Panayiotopoulos, 2001b) o hace Steve (Johnson et al., 1998).

Sea como sea, el subsistema de comunicación abstrae todo el entorno virtual a la parte *cognitiva* del agente, la responsable de decidir qué acciones realizar en él dependiendo de lo percibido.

La mayoría de los agentes pedagógicos hacen uso de todos los módulos típicos de la arquitectura básica de ITSs. Al fin y al cabo, necesitan conocer

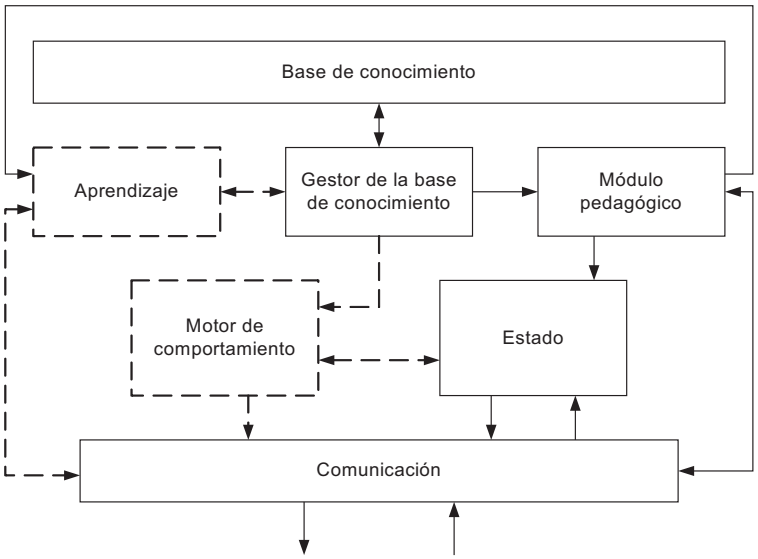


Figura 3.7: Arquitectura típica de un agente pedagógico, según Devedzic y Harrer (2005)

el conocimiento del dominio para poder enseñarlo y el modelo del usuario y distintas estrategias pedagógicas para poder amoldarse al estudiante.

En el patrón generalizado de agentes pedagógicos ("*Generalized Pedagogical Agent*", GPA) expuesto por Devedzic y Harrer (2005) estos tres tipos de conocimiento quedan agrupados en un único módulo del agente pedagógico que ellos llaman "*Knowledge Base*". En la figura 3.7, donde aparece el esquema completo del patrón con los módulos opcionales en líneas discontinuas, este conocimiento aparece bajo el nombre "base de conocimiento".

Particularmente, no estamos de acuerdo con ellos. Desde nuestro punto de vista, y después del análisis de la arquitectura de otros agentes pedagógicos y de nuestro propio sistema, el ITS *no* termina en el agente pedagógico, por lo que todo ese conocimiento debe ser externo a él, y estar disponible para el resto de elementos del sistema.

A este respecto, la arquitectura de un ITS al que se le ha añadido un agente pedagógico debe seguir, a lo sumo, la arquitectura Modelo de conocimiento–Vista descrita en la sección 3.3.2. De acuerdo a los sistemas estudiados, el agente pedagógico debe jugar el papel de Vista de al menos el modelo del estudiante y del conocimiento del dominio. Gracias a eso, su módulo cognitivo posee una *proyección* del conocimiento del ITS completo que necesita. De hecho, el GPA de Devedzic y Harrer (2005) incluye un módulo gestor de esa base de conocimiento (*knowledge base manager*), que *adapta* el conocimiento según las necesidades del resto de módulos del agente.

Por otra parte, no consideramos obligatoria la inclusión del conocimiento

de tutoría en el agente, ya que es habitual que éste incluya conocimiento específico sobre su forma de actuar. De hecho, en el GPA el módulo pedagógico aparece explícitamente fuera del módulo de la base de conocimiento, bajo el desafortunado nombre de "*problem solver*". Su misión es decidir si debe o no entrar en acción, cómo hacerlo, a qué estímulos reaccionar, etc.

El resto de módulos que componen la parte cognitiva del agente depende de las capacidades que se le quieran añadir. La mayoría tienen algún tipo de estado mental que determina sus acciones. Pueden tener un planificador para generar las acciones a realizar en un futuro y poder compararlas con las del estudiante, pueden tener un módulo encargado de generar emociones, o incluso un módulo de aprendizaje máquina que les permite modificar su conocimiento con el tiempo.

3.5.3. Ejemplo de agentes pedagógicos: Steve

En la sección 3.2.5 ya salieron algunos ejemplos de agentes pedagógicos y los relacionamos con las capacidades que tenían. En este apartado vamos a detallar la arquitectura completa de Steve ("*Soar Training Expert for Virtual Environments*"). Hemos elegido analizar Steve en vez de otro agente debido a que fue uno de los primeros agentes pedagógicos en desarrollarse, su arquitectura está muy documentada, y posee muchas de las capacidades descritas en la sección 3.2.5, en la que listábamos algunos de los tipos de interacción que permiten los agentes pedagógicos animados.

El propósito de Steve es interactuar con estudiantes en un entorno virtual inmersivo. Se ha utilizado para entrenamiento de tareas navales. Tiene capacidades pedagógicas que le permiten responder a preguntas del estudiante como "¿qué debo hacer ahora?" y "¿por qué?". También puede demostrar las acciones él mismo y usar la mirada y gestos para dirigir la atención del estudiante.

El alumno comparte el entorno virtual con el agente, de forma que mientras el usuario está interactuando con los objetos, Steve aparece a su lado observando. Cuando el usuario no sabe cómo seguir, puede preguntarle, o pedirle que acabe él la tarea. En ese caso, según va ejecutando cada acción, va explicando cada paso utilizando un generador de voz. En cualquier momento, el usuario puede pararle para preguntar las razones de una acción o para pedirle acabar por él mismo el ejercicio.

3.5.3.1. Arquitectura del sistema

El sistema consiste en varios componentes ejecutándose en paralelo (en distintos procesos e incluso distintas máquinas) conectados a través de un sistema de comunicación que distribuye los mensajes (ver figura 3.8). Resulta curioso ver que en Steve los componentes *no* son los módulos de la arquitectura clásica de ITSs que veíamos en la sección 3.3.1, sino que son los

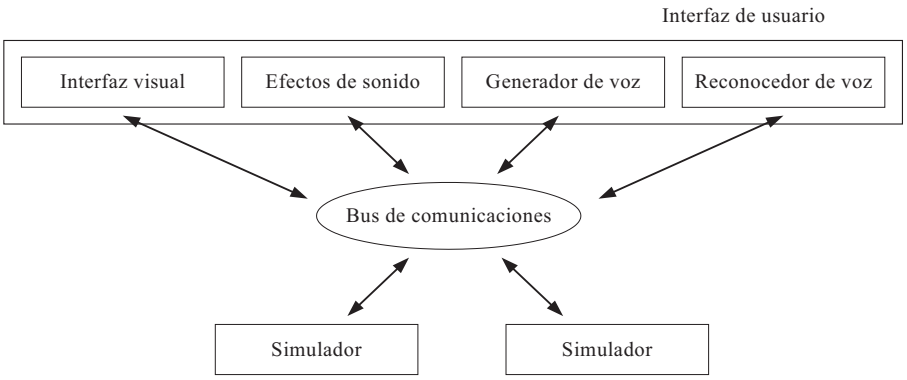


Figura 3.8: Arquitectura de Steve, según Rickel y Johnson (1999)

siguientes:

- Simulador del mundo: controla el comportamiento de los objetos del mundo virtual. Cada objeto posee un conjunto de atributos que definen tanto su apariencia física en el entorno como su comportamiento. Para cada escenario particular, se puede programar el comportamiento de cada objeto con una serie de reglas y restricciones que pueden depender de otros objetos. Puede recibir mensajes solicitando cambios de atributos, lo que puede provocar que otros objetos del entorno deban cambiar su estado. En ese caso, propaga los cambios al resto de componentes. Hay que hacer notar que el simulador se encarga únicamente de controlar el estado de los objetos, no de su apariencia.

- Interfaz del usuario: por cada componente humano existe un interfaz del usuario que permite ver y manipular el mundo virtual. Se compone de un visor a través del cual el usuario puede ver el entorno, un sistema de audio para percibir los sonidos, un generador de voz que convierte a sonido audible el texto "dicho" por el agente pedagógico, y un analizador de voz que reconoce órdenes sencillas (como "¿qué debo hacer ahora?"). Además de este analizador, el visor también puede captar la entrada del usuario para interpretar cuando quiere interactuar con alguno de los objetos virtuales.

- Agente pedagógico: Steve es considerado un componente separado. Dada su importancia, su estructura la veremos en el apartado siguiente.

Para unir a estos tres módulos existe, como dijimos, un bus de comunicación, que es capaz de enviar distintos tipos de mensajes. Cada uno de los componentes conectados a él puede declarar explícitamente estar interesado en unos tipos específicos, y el bus filtrará el resto.

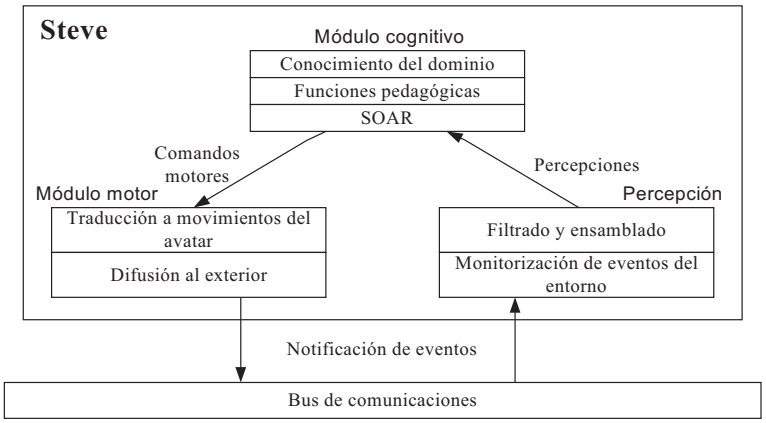


Figura 3.9: Estructura interna de Steve, según Johnson et al. (1998)

3.5.3.2. Arquitectura de Steve

Existen tres módulos distintos, el módulo de percepción, el de acción (o movimiento), y el módulo cognitivo, mostrados en la figura 3.9.

El módulo de percepción se encarga de percibir el estado del simulador. Para eso recibe los mensajes que provienen del simulador y los guarda en un conjunto de pares atributo–valor. Además, también es responsable de almacenar dónde está cada uno de los objetos del mundo así como la posición del propio avatar y del estudiante. Una última atribución a este módulo es conocer qué objetos están dentro del campo de visión del usuario. De esta forma, el módulo cognitivo podrá saber, durante las demostraciones, si el estudiante está mirando los objetos con los que él está operando.

El módulo responsable de las acciones del agente descompone en acciones primitivas los deseos del módulo cognitivo, y los envía a los visores de los estudiantes a través del bus para que estos cambien la apariencia de Steve en el visor del usuario. Las acciones recibidas desde el módulo cognitivo son por ejemplo decir una frase, moverse a un objeto o mirarlo, mover la cabeza para mostrar agrado, apuntar hacia un objeto o manipularlo. El módulo las traduce a tres tipos de mensajes: aquellos dirigidos al visor para la animación del avatar, mensajes al simulador del mundo para que cambie el estado de algún objeto, o mensajes al generador de voz de los interfaces de usuario para dar explicaciones.

El módulo cognitivo es el más importante. Está construido sobre SOAR (Laird et al., 1987) para simular el conocimiento. Dado que SOAR soporta el modelado jerárquico de tareas, es adecuado para el modelado dirigido por objetivos y comportamientos reactivo en entornos virtuales. Además, al incorporar un mecanismo de aprendizaje, permite mejorar el rendimiento del agente con el tiempo.

Utilizando el sistema de razonamiento de SOAR, se crean las capacidades pedagógicas del agente, independientes del dominio enseñado. La capa pedagógica que aparece en la figura 3.9 crea un lenguaje específico sencillo (que posteriormente traduce a reglas SOAR), con el que se puede definir el conocimiento del dominio, que en el caso de Steve es el modo en el que hay que ejecutar cada una de las tareas en el simulador.

Esas tareas se representan como una jerarquía de planes; cada tarea es un conjunto de pasos que pueden ser o acciones primitivas o acciones compuestas, responsables de la estructura jerárquica. Entre los distintos pasos de una tarea, pueden existir restricciones de orden, definiendo un orden parcial entre los pasos. Además, cada paso posee información relativa al objetivo del paso dentro de la tarea.

Cuando el módulo cognitivo utilizando el motor de razonamiento de SOAR y la capa pedagógica planifica una tarea, es capaz de tener en cuenta las restricciones anteriores, y generar las explicaciones para cada paso si es preguntado.

Resumen

Después de ver en el capítulo anterior la forma en la que se estructuran los videojuegos, este capítulo se centra en las arquitecturas de los programas educativos.

Después de una breve introducción histórica y una descripción de las ventajas de utilizar entornos de aprendizaje inmersivos, en la sección 3.3 hemos analizado los componente más comunes que presentan los sistemas de tutoría inteligentes. En esta sección hemos descrito tres arquitecturas típicas: la arquitectura clásica, la basada en la pareja modelo de conocimiento–vista, y por último la basada en agentes.

Dado que los sistemas educativos que perseguimos tienen la capacidad de albergar agentes pedagógicos, hemos proseguido nuestro análisis inspeccionando el tipo de problemas y soluciones encontradas en sistemas que presentan un entorno virtual inteligente al que se ha añadido un agente virtual.

La sección 3.5 termina el capítulo centrándose en los sistemas de tutoría inteligentes que han sido mejorados añadiendo algún agente pedagógico.

Con estas bases, los capítulos siguientes definirán una metodología y una arquitectura que aune ambos mundos, para conseguir un modo eficiente de implementar sistemas educativos basados en videojuegos.

Notas bibliográficas

Dentro de la literatura sobre software educativo, el libro Wenger (1987) se ha convertido en un clásico en el mundo de los ITSs, al ser el primero que

estableció lo que en la sección 3.3.1 hemos llamado *arquitectura clásica de ITSs*.

En cuanto a artículos relacionados con entornos virtuales inteligentes, un buen punto de partida son las actas de las distintas ediciones del congreso Intelligent Virtual Agents de periodicidad bianual, en años impares. Algunas de las referencias incluidas en este capítulo han sido publicadas en él.

Relacionado con ITSs existe otro congreso bianual, esta vez en años pares, "Intelligent Tutoring Systems", en el que aparecen artículos relacionados con cada uno de los módulos de la arquitectura explicada en este capítulo, incluidos los agentes pedagógicos. Tampoco hay que olvidar el congreso internacional celebrado cada dos años, "International Conference on Artificial Intelligence in Education", que celebró su decimotercera edición en 2007, y la revista "International Journal of Artificial Intelligence in Education", en marcha desde 1989.

Relacionado con el módulo experto de la sección 3.2.1 y las distintas posibilidades de expresar ese conocimiento, se puede consultar Gómez Martín, Gómez Martín, González Calero y Jiménez Díaz (2005b), en el que se analizan las repercusiones de esta representación en el sistema completo.

Por último, relacionado con la mejora de los sistemas educativos incluyendo un agente pedagógico dentro de una historia interesante comentado en la sección 3.2.5, se puede consultar Gómez Martín et al. (2004b).

Capítulo 4

Una metodología para la creación de contenidos dirigidos por datos

Comienza por el comienzo – dijo, muy gravemente, el Rey –, y sigue hasta que llegues al final; entonces paras.

Lewis Carroll, Alicia en el país de las maravillas

En este capítulo se presenta la metodología de desarrollo de las aplicaciones educativas descritas en los capítulos anteriores. Para eso, primero describimos a alto nivel las fases que deben seguirse durante su creación y las herramientas de soporte necesarias. También describimos nuestra propuesta de metodología para la creación de los contenidos que presenta la aplicación. Por último, el capítulo describe dos técnicas de uso del análisis formal de conceptos para el análisis de los contenidos creados.

4.1. Introducción

Tras dos capítulos en los que hemos visto las ideas generales del desarrollo de software de entretenimiento y de las aplicaciones educativas, este cuarto capítulo se centra en una propuesta metodológica para el desarrollo de sistemas educativos basados en videojuegos, centrándonos en la generación del contenido.

Lo primero que haremos es ver en qué fases está dividido el proceso de creación de una de estas aplicaciones. El trabajo en estas fases, como veremos, involucra tanto trabajo de programación como de creación de contenido lúdico y educativo. La división en fases, de hecho, la marca la evolución de este último aspecto, por lo que, a pesar de existir fases, no debe entenderse que

la programación sigue una metodología *en cascada*. Ésta puede evolucionar de manera independiente al contenido, utilizando cualquier metodología de ingeniería del software, como el proceso unificado o las distintas metodologías ágiles.

Dado que precisamente es el contenido del videojuego educativo la parte más importante, nuestra propuesta metodológica se centra en determinar el mejor método para su creación. Como veremos, cuando hablamos de estos contenidos nos estamos refiriendo tanto a la forma del entorno virtual donde se desarrolla la acción como a los ejercicios y conceptos que están siendo enseñados.

Posteriormente, realizamos un estudio sobre las herramientas que pueden utilizarse para dar soporte al desarrollo de este tipo de aplicaciones. En particular, describimos muchas de las utilidades que suelen ir asociadas a desarrollos software, analizando las distintas alternativas existentes para la implementación de videojuegos educativos. Por último, el capítulo termina con la descripción de dos técnicas que hacen uso de Análisis Formal de Conceptos para ayudar a comprobar si los contenidos creados con la metodología propuesta son suficientes y si el nivelado del videojuego es correcto.

4.2. Proceso de creación de un videojuego educativo

Los videojuegos son obras de creación, que exigen la combinación de talento, técnica y creatividad (Arévalo, 2005). Por lo tanto, su creación involucra, a grandes rasgos, la mezcla de dos procesos, el artístico y el técnico. El primero es el responsable de los contenidos que aparecen en pantalla, la definición de la jugabilidad, estructura de los mundos virtuales en que nos encontramos, etc., mientras que el segundo es el que desarrolla la tecnología (*software*) para mostrar esos contenidos. Si añadimos a los videojuegos la componente educativa, a este desarrollo hay que añadirle la creación de los contenidos educativos que se pretenden enseñar.

Por lo tanto, durante el proceso de creación de estas aplicaciones conviven tres mundos bien distintos:

- *Componente artístico*: este componente suele entenderse como el conjunto de elementos *estéticos* que aparecen en el entorno virtual, como personajes o texturas. No obstante, también hay aquí que añadir la concepción del juego, la definición de la jugabilidad, guión y estructura de los entornos virtuales. De hecho, en la década de los 80, Crawford en su libro sobre el diseño de juegos, no dedicaba una atención especial al primer tipo de contenido, y se limitaba a destacar la importancia del segundo.

- *Componente pedagógico*: es el responsable de decidir, por ejemplo, de

qué forma se van a presentar los contenidos o conceptos a enseñar, cuál es la forma más efectiva de hacerlo para facilitar el aprendizaje, o el orden en el que se llevará a cabo.

- *Componente técnico*: es la parte responsable de lidiar con la máquina en la que se va a ejecutar la aplicación, creando tanto las herramientas necesarias para ayudar a la construcción de los componentes anteriores, como la aplicación que va a hacer uso de ellos.

La creación del contenido artístico y pedagógico está, por lo tanto, condicionada a la forma en la que se programe la aplicación que utiliza ese contenido. Las secciones 4.3 y 4.4 detallan las alternativas existentes en videojuegos educativos, así como nuestra propuesta metodológica.

Las fases comúnmente aceptadas en el desarrollo de un videojuego son las siguientes:

- *Concepto*: en esta fase se define la idea fundamental del juego. Empieza con una simple frase que se va refinando poco a poco. El grado de detalle alcanzado, no obstante, no es excesivo, para no cerrar posibilidades futuras poniendo detalles u opciones secundarias.

- *Preproducción*: una vez definido el concepto, el siguiente paso es refinarlo, haciendo una investigación detallada sobre el tema elegido, para tener al menos un esbozo del guión y contexto histórico (si lo hay). Es en esta fase cuando se decide el aspecto visual que va a tener la aplicación. Desde el punto de vista técnico, en esta fase se realiza un análisis de las librerías externas (o COTS, ver sección 2.3.1) que se utilizarán para el desarrollo. También es el momento de implementar distintos prototipos que ayudarán en la elección de esas librerías, y que valdrán como prueba de que las ideas de diseño creadas en la etapa anterior funcionan desde el punto de vista del entretenimiento.

  Al final de la fase de preproducción, se habrán desarrollado todas las piezas fundamentales de la aplicación final, como las herramientas de generación de contenidos o exportadores, el motor gráfico o la parte básica de la inteligencia artificial.

- *Producción*: en esta etapa, el equipo de desarrollo alcanza el tamaño máximo. Es cuando se desarrollan todos los contenidos que llevará la aplicación final. Eso requiere un amplio equipo de artistas y desarrolladores de niveles, que generan los componentes visuales (modelos) y los mapas o entornos en los que se desarrolla la acción. El equipo de programación se encarga, de manera paralela, de refinar las herramientas creadas en la etapa anterior, para amoldarlas a las nuevas necesidades que van surgiendo en su uso diario. También se encarga de la creación

de los comportamientos, gestión de entidades, etc., que irán en el juego. Al final del proceso, se tiene una versión *completa* del juego, con todos los menús, niveles, misiones y posibles traducciones.

- *Control de la calidad*: en esta fase, previa al lanzamiento final del juego, se prueba intensivamente la aplicación en busca de errores. Para ello gente tanto externa como interna al equipo de desarrollo comprueban si el juego funciona en todas las condiciones de ejecución, en todas las plataformas y a lo largo de todos los niveles. Es normal que algunos de los errores detectados *no* se resuelvan por falta de tiempo. Al final de esta fase, se consigue la versión del juego que llegará a los compradores.

- *Mantenimiento y explotación*: a pesar de que el desarrollo se hace intentando asegurar la ausencia de errores, una vez terminado el juego, siempre se detectan algunos. En esta fase, *posterior* a la salida al mercado, puede ser necesario sacar actualizaciones para arreglar tanto los problemas detectados posteriormente como los que quedaron sin resolver en la fase anterior al lanzamiento. En juegos multijugador masivos con mundos persistentes, que dependen de la existencia de un servidor para almacenar los datos, la etapa de mantenimiento es mucho más larga que en otros juegos.

Para soportar todo este proceso, en un estudio de desarrollo hace falta una estructura de equipo clara. Son necesarios los departamentos tradicionales de dirección, gestión, recursos humanos y marketing, todos ellos "externos" al desarrollo de la aplicación final. Los departamentos que realmente implementan las fases anteriores son (i) el departamento de programación, encargado de la creación del motor de juego/tecnología, herramientas y lógica del juego, (ii) el departamento gráfico, para todo el diseño artístico, modelado, texturado y animación, y (iii) el departamento de diseño, que escribe y refina el documento que define los elementos interactivos y narrativos, reglas, mecánicas y misiones.

Existen dos departamentos adicionales que también intervienen en las fases anteriores, pero que habitualmente se consideran departamentos *transversales* que trabajan simultáneamente en todos los proyectos del estudio. El primero de ellos es el departamento de sonido, que se encarga de diseñar el estilo y ambiente sonoro, componer las músicas y gestionar su grabación; también es responsable del resto de efectos sonoros y voces. Un último departamento es el responsable de las pruebas, conocido como el departamento de *QA* (del inglés, *quality assurance*).

En realidad, aunque se defina explícitamente una fase para pruebas, éstas tienen lugar a lo largo de todo el proceso de desarrollo. La más importante es la comprobación de si el videojuego de verdad *entretiene*. Y dada su importancia, esta comprobación se realiza desde el principio del proceso. Por

ejemplo, Crawford (1984) comentaba que del casi centenar de ideas de juego que había tenido hasta ese momento (fase de concepto), había explorado menos de 40, y únicamente ocho habían terminado en un juego desarrollado[1]. Hoy día después de la fase de preproducción se prueba la jugabilidad utilizando prototipos, y se vuelve a estudiar la viabilidad del proyecto antes de entrar en la producción. Una vez comenzada ésta, las evaluaciones sobre la componente lúdica del juego continúan, y pueden llevar a cancelaciones de proyectos tardías si no se superan.

Si volvemos a las aplicaciones educativas sin componente lúdico (es decir, que no pueden ser consideradas videojuegos), podemos hacer una analogía en el proceso de desarrollo:

- *Concepto*: consiste en establecer qué es lo que se quiere enseñar. Dado que normalmente las aplicaciones educativas se realizan "por encargo", la elección del tema principal de la aplicación vendrá impuesta de antemano.

- *Preproducción*: igual que en la construcción de un videojuego, en este momento el equipo puede plantearse, si no viene impuesto por el cliente, el tipo de aplicación que se construirá. Dependiendo de la decisión tomada, se analizarán herramientas ya disponibles o se construirán herramientas propias para el desarrollo de la aplicación completa. Por ejemplo, si se decide enmarcarse dentro del área de *enseñanza asistida por ordenador* (CAI) que nombrábamos en la sección 3.2, el equipo puede decantarse por utilizar una herramienta ya consolidada como Macromedia AuthorWare (Adobe, 2006). Para otros desarrollos se puede plantear la creación de herramientas propias. En esta fase se debe decidir con detalle de qué forma se van a presentar y a secuenciar los contenidos. Además, si, por ejemplo, la aplicación educativa va a consistir en un simulador, habrá que desarrollar, al menos, la parte básica de éste.

[1] El proceso de desarrollo presentado por Crawford (1984) no es exactamente como el expuesto aquí; en su caso, definía seis fases: concepto, preproducción, diseño, pre-desarrollo, desarrollo, control de calidad y post-mortem. Como se puede ver, presta especial atención al diseño del juego, al que dedica hasta tres fases antes de que el trabajo de programación empiece. En particular, su primera fase consistía en elegir el objetivo y ambientación o tema del juego ("*Choose a goal and a topic*"), que podemos equiparar con nuestra fase de concepto. En las dos fases siguientes se realiza una investigación y preparación sobre el tema y el diseño del juego, y no es hasta la cuarta fase cuando se realiza el diseño software (fase de pre-desarrollo) y programación (fase de desarrollo). En nuestro modelo, tanto las dos fases específicas de diseño como las dos fases relacionadas con la programación son ejecutadas en paralelo en las fases de preproducción y producción. Sus dos últimas etapas son la control de calidad, y *post-mortem*. El significado de esta última fase no es el que se entiende hoy en día, sino que consistía simplemente en ver cómo evolucionaba el juego en el mercado, las opiniones de los críticos especializados y público en general.

- *Producción*: es la fase en la que se sitúa el grueso de la creación de contenidos. Si las herramientas utilizadas tienen la madurez suficiente, podrán utilizarse directamente, y la fase se limitará a la creación de esos contenidos. Si las herramientas han sido desarrolladas específicamente para la aplicación, es posible que requieran cambios.

- *Evaluación de la aplicación*: la evaluación en este caso es doble. Hay que comprobar (i) que la aplicación funciona en todos los casos, (ii) el propio *impacto educativo* en los estudiantes (Mark y Greer, 1993; Inoue, 2001).

- *Mantenimiento y explotación*: dependiendo de la aplicación concreta la etapa posterior al lanzamiento será más o menos importante y larga. Si la aplicación genera registros durante la ejecución, se pueden analizar estos para mejorar futuras versiones de la misma aplicación o familia de aplicaciones. También puede ser necesario revisar contenidos o arreglar errores.

Algunos de los profesionales que eran necesarios para el desarrollo de videojuegos son también requeridos en la creación de aplicaciones educativas. Por ejemplo, se utilizan programadores para desarrollar herramientas de creación de contenidos o para programar extensiones (*plug-in*s) o retoques de las mismas. Si existe un componente de contenido multimedia, también se necesitarán grafistas y expertos de sonido.

Sin embargo, el departamento más importante es el de expertos del dominio que son los que conocen el área que se pretende enseñar. Éstos son capaces de describir el contenido, y a veces incluso generarlo con las herramientas utilizadas en el desarrollo. Un equipo de psicólogos y pedagogos, además, se asegura de que el orden en esa presentación sea idóneo para su asimilación, al presentar una progresión de dificultad adecuada.

Una vez detallados los procesos de desarrollo de videojuegos y aplicaciones educativas por separado, podemos destacar tres diferencias fundamentales:

- La cantidad de esfuerzo en programación requerida: normalmente, los videojuegos requieren mucho más trabajo de programación que las aplicaciones educativas, en donde lo más importante (especialmente en los CAI) es la generación del contenido didáctico. En muchas ocasiones, en la fase de "producción" de la aplicación educativa no es necesario programar, y se limita únicamente a crear esos contenidos.

- Los objetivos y momentos de la evaluación de la aplicación: el objetivo más importante de la evaluación de una aplicación educativa es comprobar si ésta realmente sirve para enseñar al usuario/estudiante, por lo tanto debe hacerse principalmente una vez que ésta está creada.

En un juego como hemos dicho anteriormente, la evaluación realizada durante y después de la producción sirve principalmente para validar el buen comportamiento en ejecución, pero la comprobación de si éste *entretiene* se ha realizado mucho antes.

- Los profesionales necesarios: en ambos procesos son necesarios programadores y grafistas. Sin embargo, el papel del diseñador del juego que detalla los niveles y misiones de un videojuego es sustituido por el del experto del dominio que define los contenidos a presentar en la aplicación educativa[2]. El equipo de QA que en un videojuego comprueba si la dificultad se incrementa de manera gradual es sustituido por psicólogos y pedagogos que hacen lo propio con los contenidos educativos.

El proceso de creación de un videojuego educativo pasa por *fusionar* ambos procesos en uno solo. En los siguientes apartados analizamos cada uno de los pasos.

4.2.1. Concepto

En esta primera fase, los objetivos siguen siendo los mismos que en el caso del desarrollo separado: definir a grandes rasgos qué tipo de aplicación se desea hacer. Para eso, hay que cubrir dos aspectos:

- Por un lado se debe seleccionar el *objetivo* del videojuego educativo, es decir, qué pretendemos enseñar con él. En el caso de una empresa dedicada a la construcción de estas aplicaciones, es muy posible que el objetivo venga impuesto por el cliente, aunque puede que la decisión sea de la propia empresa si decide crear un videojuego en vistas a un cliente potencial específico. Esta parte se corresponde con el objetivo de la fase de concepto en el desarrollo de las aplicaciones educativas.

- Por otro lado debe seleccionarse la *ambientación*, que va a *envolver* el objetivo anterior en una componente lúdica. Es decir, debemos cubrir los objetivos marcados para la fase de concepto del desarrollo de un videojuego.

Es destacable que el propio Crawford (1984) hace una distinción similar cuando habla de la primera fase en el diseño de un videojuego. Indica que en esa fase hay que elegir un objetivo y un tema, y pone el ejemplo del juego Eastern Front 1941, en el que el objetivo estaba relacionado con la naturaleza de las reglas y la diferencia entre la potencia bélica y la efectividad, mientras que el tema o ambientación era la guerra entre Rusia y Alemania.

[2]En un videojuego, el diseñador suele *documentarse* para conocer con cierto grado de profundidad el *área* en el que se desarrolla el videojuego, pero nunca llega a convertirse en un experto del dominio.

4.2.2. Preproducción

Una vez determinados los objetivos básicos y la idea del componente lúdico, la preproducción es el primer paso de desarrollo.

En esta fase, el objetivo y la ambientación se combinan para crear el juego educativo. Esa unión determinará el éxito o el fracaso de la aplicación. Una combinación bien hecha *difumina* la separación entre ambos aspectos, por lo que aprovecha al máximo la componente de motivación del juego mientras conserva la comunicación efectiva de los contenidos pedagógicos. Por tanto, requiere la participación tanto de expertos en el diseño de juegos como de pedagogos o psicólogos y expertos en el dominio a enseñar.

En muchos juegos educativos se utiliza la *simulación* de los entornos reales en los que se desarrollan las acciones que se quieren enseñar. Así es por ejemplo Tactical Iraqi (Johnson et al., 2005). En contextos educativos donde la simulación real del entorno no es posible, se puede hacer uso de metáforas (Gómez Martín, Gómez Martín, González Calero y Palmier Campos, 2007d), como en JV^{2}M que veremos en el capítulo 6 o Dimenxian[3]. En cualquier caso, las decisiones tomadas en este punto determinarán el modo en el que se crearán los contenidos, que trataremos en secciones posteriores.

El resultado de todas estas decisiones terminará con la escritura del *documento de diseño* del juego educativo, que será el eje conductor de todo el proceso de creación. Esta piedra angular, también existente en la construcción de videojuegos tradicionales, contiene una descripción de lo que debería ser la aplicación final. Para ello, debe al menos cubrir los siguientes puntos:

- Una introducción que destaque los puntos fuertes y originales de la aplicación.
- El método en el que se va a llevar a cabo la integración entre la parte lúdica y la parte de contenido educativo, que ya hemos comentado.
- Una descripción de interacción con la aplicación final, para que el lector entienda cómo va a ser su funcionamiento general.
- Una lista de las características más importantes, principalmente desde el punto de vista del desarrollo. Por ejemplo, indicar si la aplicación se va a desarrollar on-line o qué tipo de controladores va a manejar el usuario.
- Algún boceto del aspecto final que el diseñador tiene en mente.

Durante la preproducción también se empieza a desarrollar parte de la aplicación final. Dependiendo del formato de juego elegido, es posible que

[3] Juego para aprender álgebra desarrollado por Tabula Digita (`http://www.dimenxian.com/`).

algunas de las herramientas necesarias para la elaboración del contenido ya estén disponibles; en caso contrario, es en este momento cuando se crean.

En esta fase, también se importa del desarrollo de los videojuegos la necesidad de pruebas de concepto del diseño por medio de prototipos. De esta forma, se puede averiguar lo antes posible si el videojuego educativo va a resultar atractivo a los jugadores/estudiantes o no. En caso negativo, se deberán replantear las decisiones tomadas en la fase de concepto (lo que llamábamos la ambientación) y el documento de diseño donde se fijaba la unión entre ella y los contenidos pedagógicos.

Al final de la fase de preproducción, debe existir una versión ejecutable de la aplicación que dé una idea bastante fiel del aspecto que tendrá al final.

4.2.3. Producción

En esta fase es donde se realiza el verdadero desarrollo de la aplicación. Utilizando como base el resultado de la fase anterior, se construye toda la *masa* de contenidos, que en este caso consistirá no sólo en la parte visual y de interacción ligadas fundamentalmente a la componente lúdica, sino también el contenido educativo.

En esta fase habrá irremediablemente que afinar las herramientas realizadas en la etapa anterior, y se programará el comportamiento de todos los personajes (inteligencia artificial), niveles, unidades y mecánicas.

Igual que en el desarrollo de videojuegos, tanto durante esta fase como en la anterior, se realizan pruebas sistemáticas de la aplicación. Los diseñadores deben evaluar continuamente si el diseño creado es entretenido, y en caso negativo, tomar medidas. Esas medidas pueden ser modificaciones sencillas en las mecánicas y niveles, o pueden suponer empezar desde el principio por la ambientación.

En algún momento de la fase de producción se consigue una versión en la que todo el esfuerzo de programación ha terminado, y lo único que queda es alimentar a ese motor con contenidos en forma de datos, gracias a las arquitecturas dirigidas por datos (ver sección 2.4). Una vez completado el contenido, se da por concluida la producción, y se pasa a la siguiente fase.

4.2.4. Control de calidad

Una vez dado por concluido el desarrollo del videojuego educativo, se da paso a una fase en la que las pruebas se intensifican, y el desarrollo que se hace es meramente para corregir los errores detectados. La fase suele llamarse fase de QA[4], o lo que es lo mismo validación o garantía de calidad.

El equipo de QA en realidad está activo desde las fases iniciales del desarrollo, ejecutando el producto parcial en busca de fallos, tanto de ejecución

[4]Como ya se ha dicho, se corresponde a las siglas inglesas de *quality assurance*

como de discrepancia con los documentos de diseño. Cuando durante estas fases detectan algún error, informan de él al equipo de desarrollo por medio de herramientas de seguimiento de errores (descritas en la sección 4.5.4).

En esta última fase, una vez dado por concluido el desarrollo completo, el único esfuerzo de programación que se hace es aquel encaminado a la corrección de los errores, por lo que el número de programadores se reduce significativamente.

En este momento, es el equipo de QA en el que recae más responsabilidad. La versión completa (conocida como versión *beta*), se distribuye entre personas *externas* al equipo de desarrollo (conocidos como *betatesters*, o probadores de versiones *beta*). El método idóneo de funcionamiento es que estas personas externas se comuniquen únicamente con *el equipo de QA*, mediante una herramienta de gestión independiente de la utilizada internamente entre el equipo de QA y el de desarrollo. El equipo de QA se encarga de clasificar, eliminar duplicados e incluso filtrar e ignorar los errores que llegan del exterior, e informar al equipo de desarrollo de todos aquellos que deberían ser arreglados.

Durante esta fase también se deben hacer las últimas pruebas para *equilibrar* o *nivelar* la componente lúdica y pedagógica de la aplicación.

Con respecto al nivelado de la componente lúdica, nos referimos a hacer que los retos propuestos por la aplicación se ajusten al nivel del usuario, ya que los juegos más divertidos son aquellos jugados entre dos oponentes de un nivel similar[5] (Graepel et al., 2004). Estas pruebas de nivelado del juego se habrán hecho también en la fase de preproducción mediante prototipos, y en la fase de producción con versiones intermedias del videojuego final, pero en este momento pueden utilizarse jugadores *reales* externos al equipo de desarrollo. En la sección 4.6.3 veremos cómo podemos utilizar análisis formal de conceptos para analizar los registros generados de las partidas de estos jugadores para sacar conclusiones útiles para el nivelado.

Por último, respecto a la componente pedagógica, debemos comprobar si los contenidos educativos creados son suficientes o no. Gracias a la separación entre la parte lúdica y la educativa de los contenidos que detallamos en las secciones 4.3 y 4.4, veremos que en esta fase podremos también comprobar si los contenidos educativos son suficientes utilizando análisis formal de conceptos.

[5]Resulta curioso que esto ocurra incluso en la novela "2001, una odisea en el espacio", donde el autor indica que HAL, el ordenador de a bordo podía jugar contra los astronautas a varios juegos como ajedrez y damas, pero que "se dejaba ganar" la mitad de las veces, para no bajar la moral.

4.2.5. Mantenimiento y explotación

Igual que en el caso anterior, las tareas de mantenimiento que requiere un videojuego educativo son las suma de las que requiere una aplicación educativa y un videojuego por separado.

En particular, es posible que se requieran actualizaciones para solucionar errores detectados después de la publicación de la aplicación, o actualizar algunos de los contenidos educativos.

Para permitir actualizaciones, el programa ha debido ser implementado prestando especial atención a la posibilidad de tener ficheros de datos con distintas versiones. En aplicaciones en red, además, los protocolos de comunicación también deben ser conscientes de esa posibilidad, y negar la comunicación entre dos aplicaciones con actualizaciones distintas.

4.3. Creación de contenidos

La arquitectura dirigida por datos que comentábamos en la sección 2.4 establece una separación entre el motor del juego y los datos o contenido del mismo. Las acciones que se realizan tanto en la fase de preproducción como de producción están claramente marcadas por este tipo de arquitectura. Desde el punto de vista técnico, el estudio dedica un esfuerzo considerable a mantener un motor de juego centralizado, controlado por unos pocos programadores, mientras el resto es dedicado a la creación de código específico de esa aplicación concreta (como el comportamiento de NPCs) y a la creación de herramientas para la generación de contenidos para ese motor.

El resto de personas se dedica a la *generación de contenidos* de la aplicación utilizando esos editores. Para eso se establece un mecanismo de gestión y control de esos contenidos, estableciendo las fases por las que van pasando los recursos, de tal forma que a las personas que los crean les sea posible ver fácilmente en el motor del juego (resultado final) lo que está editando en la herramienta.

Los buenos resultados de las arquitecturas dirigidas por datos son aprovechados al máximo cuando el género de la aplicación está *consolidado*, es decir, cuando las mecánicas, formas de interactuar con el usuario y los tipos de contenidos están claros y no sufren modificaciones entre aplicaciones. En esos casos, el motor de la aplicación puede cubrir toda la funcionalidad necesaria para cualquier entrada de datos, de tal forma que estos se convierten en la única diferencia entre distintas aplicaciones.

En las siguientes dos secciones describimos el modo de generar contenidos para aplicaciones educativas por un lado y para videojuegos por otro. Por último, en la sección 4.3.3 describimos las distintas alternativas de creación de contenidos que soporta nuestra propuesta de arquitectura de videojuegos educativos.

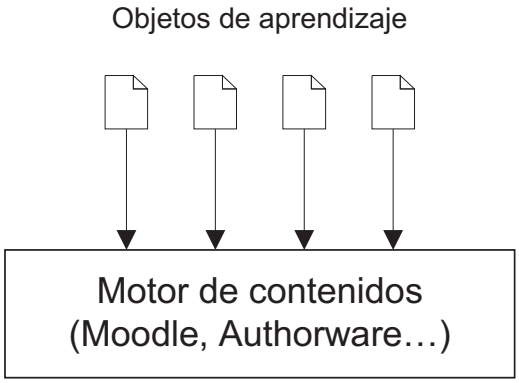


Figura 4.1: Esquema de contenidos en aplicaciones educativas

4.3.1. Contenidos en aplicaciones educativas

No cabe duda que lo más importante de una aplicación educativa es el conocimiento del dominio (Devedzic et al., 1998), por lo que el esfuerzo a la hora de su construcción se centra, precisamente, en la creación del contenido relacionado con éste.

Si nos centramos en los aplicaciones de enseñanza asistidas por ordenador (CAI, ver sección 3.2), desde los primeros productos se ha intentado independizar el motor de visualización de los propios datos, de tal forma que por un lado exista el motor que muestra los contenidos, y por otro lado estén los datos, en ficheros externos (ver figura 4.1). Uno de los primeros sistemas que lo hicieron fue PLATO (Wikipedia (PLATO)). Un ejemplo más reciente son las aplicaciones desarrolladas utilizando Authorware (Adobe, 2006), que permite editar el contenido textual y añadirle otros recursos multimedia, como imágenes y sonidos.

Hoy por hoy, sin embargo, los ejemplos más representativos son aquellos contenidos desarrollados en SCORM (Advanced Distributed Learning (ADL), 2004), que es el estándar más importante para la creación de contenidos de aprendizaje (u *objetos de instrucción*) reutilizables en distintos frameworks o motores de aprendizaje basados en Web. Una de las consecuencias más importantes de la existencia de este estándar es que ha permitido la creación de contenidos de manera independiente a la aplicación utilizada para su visualización. Los principales beneficiados han sido los sistemas gestores de aprendizaje o LMS (abreviatura de la denominación inglesa, *Learning Management System*). Estos *motores*, como Moodle (Dougiamas y Taylor, 2003), permiten visualizar los contenidos definidos en SCORM en un navegador Web, y monitorizar al alumno durante todo el proceso de aprendizaje.

Otro de los puntos importante, además del propio contenido o exposición

de los temas a enseñar, es la existencia de ejercicios o preguntas, que son las que (idealmente) guían el proceso de aprendizaje. SCORM también permite incluir estos ejercicios o tests, y determinar, en base a sus resultados, los siguientes contenidos educativos a presentar.

Si pensamos en aplicaciones educativas más sofisticadas que no se limiten a la presentación casi secuencial de contenidos sino que incorporen módulos inteligentes, se necesitan otros tipos de contenidos más avanzados. Por ejemplo, el conocimiento del dominio puede venir dado por un conjunto de reglas, ontologías que mantienen los conceptos a enseñar o una base de ejercicios. En esos casos, la aplicación que visualiza los contenidos posee un *motor de razonamiento* que aprovecha esas reglas, ontologías o ejercicios para presentar explicaciones, seleccionar el contenido más adecuado a ser presentado o resolver los ejercicios propuestos.

4.3.2. Contenidos en videojuegos

En el área de los videojuegos, el género que tradicionalmente mejor ha aprovechado la arquitectura dirigida por datos ha sido el de la narración interactiva (dos casos particulares se detallaban en las secciones 2.7.3.1 y 2.7.3.2). Por ejemplo, LucasArts creó numerosas aventuras gráficas utilizando primero su motor SCUMM y después GrimE, a los que hacían muy pocas modificaciones entre título y título[6].

Los datos en este caso son tanto la parte gráfica como los *scripts* que definen la parte lógica o de comportamiento de los personajes que aparecen. Por ejemplo, en estos ficheros de texto se definen los árboles de conversación o las repercusiones que tienen las acciones del usuario sobre el entorno.

Para el resto de géneros, las arquitecturas dirigidas por datos son, no obstante, también perfectamente aplicables. La única diferencia es que en este caso es mucho más complicado aprovechar sin retocar los motores para varios títulos, ya que los géneros cambian las mecánicas introduciendo innovaciones no soportadas por las versiones anteriores de los motores.

A continuación aparece una lista de los contenidos (o datos) más importantes necesarios para la creación de un juego:

- Recursos gráficos: tanto los de dos dimensiones para el interfaz del usuario (menús y HUD) como los modelos en tres dimensiones, necesarios para caracteres, entorno estático y objetos. Para su creación, existen gran cantidad de herramientas de modelado, como 3DStudio Max o Alias Wavefront.

[6]El hecho de que las aventuras gráficas se presten tanto a crearse únicamente con ficheros de datos ha dado lugar a investigaciones que mezclan aplicaciones de e-learning tradicionales con aventuras gráficas 2D, donde cada aventura (representada con ficheros de datos) representa un objeto de aprendizaje (Martínez-Ortíz et al., 2006; Moreno-Ger et al., 2006).

- Mapas: son los más importantes, ya que guían la carga del resto de recursos. Los mapas determinan qué entidades aparecen en el entorno virtual y, por tanto, qué recursos (gráficos, sonidos y scripts) será necesario cargar. Para su edición, se necesitan herramientas especializadas, normalmente creadas expresamente para el juego particular.

- Scripts: como decíamos en la sección 2.7, los scripts son parte del código que define el comportamiento del juego, definido utilizando un lenguaje de programación que se interpreta en tiempo de ejecución. Las herramientas necesarias para su creación son principalmente editores de texto y un buen entorno para la depuración que, lamentablemente, no siempre existe.

- Sonidos: el componente sonoro de un juego está recibiendo cada vez más atención por parte de los estudios. No sólo nos estamos refiriendo aquí a las grabaciones de voces en *off* o personajes, sino también a la música de ambiente y efectos.

Los mapas son la representación más clara de las ideas que el diseñador ha plasmado en el documento de diseño. Son ficheros de datos que contienen la lista de todas las entidades (u objetos del juego, ver sección 2.6) que aparecen en ese nivel. Opcionalmente, pueden contener también parte de los modelos o geometría de ese nivel (típicamente la geometría estática, como paredes o mobiliario con el que no se puede interactuar).

La principal característica de los mapas son las entidades que los diseñadores de niveles colocan en el entorno. Cada una de esas entidades tiene un comportamiento perfectamente definido, en parte por el propio motor del juego y en parte por el código creado en los scripts. La función del diseñador de niveles es "traducir" el funcionamiento del nivel indicado en el documento de diseño a una serie de entidades colocadas estratégicamente y con unas propiedades escogidas de tal forma que su comportamiento en ejecución cuadre con el indicado en el documento. Podemos pues decir que el diseñador de niveles "cablea" el documento de diseño en un mapa (Eladhari, 2002).

En las aventuras conversacionales, esa traducción es más o menos sencilla, gracias al grado de refinamiento conseguido por los motores de juegos dedicados a estos géneros. Aunque la traducción no es directa, es relativamente sencillo pasar del documento de diseño a los contenidos, por medio de variables que definen condiciones del juego, o árboles de conversaciones.

En juegos más complicados, la traducción es más compleja. En esos casos, el diseñador tiene una batería de entidades *generales* que debe ver como componentes atómicos con los que conseguir el comportamiento descrito en el documento de diseño. Algunas de las entidades aparecerán como tal en ese documento (como los distintos enemigos o armas), mientras que otras estarán pensadas para permitir esa traducción. Ejemplos de este último tipo

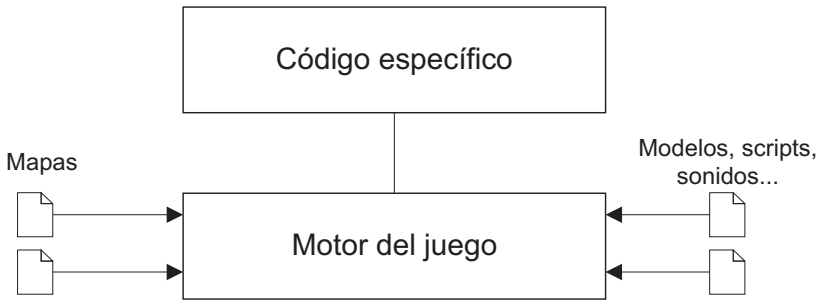


Figura 4.2: Esquema de contenidos en juegos

de entidades especiales son los disparadores de acciones (en inglés *trigger*), utilizadas para que se ejecute una acción cuando, por ejemplo, el jugador colisiona con ellos, o las anclas (en inglés *anchors*), que se utilizan como entidades invisibles utilizadas por la inteligencia artificial para encontrar localizaciones especiales, como el sitio en el que colocarse para manipular la rueda de un coche (Martel, 2006).

La figura 4.2 muestra un esquema a alto nivel de la estructura de un juego dirigido por datos. El motor del juego es la parte que puede aprovecharse entre distintos títulos, mientras que el código específico está desarrollado expresamente para el juego en cuestión (comportamiento de personajes y otros objetos del juego). El mapa es cargado por *el motor del juego* que utilizará alguna de las distintas técnicas existentes para la creación de las entidades definidas en el código específico (ver sección 2.6.2). Este esquema deja claro que el formato de los mapas *depende del motor del juego* y no del código específico.

4.3.3. Contenidos en videojuegos educativos

Los videojuegos educativos presentan, en un entorno virtual en forma de juego, una serie de contenidos educativos. Dependiendo del tipo de juego elegido, es decir, dependiendo de la *ambientación* o forma de presentar los contenidos (ver sección 4.2.1), éstos serán textos o explicaciones en el sentido estricto de la palabra, o estarán integrados entre las entidades del juego, que los presentarán en los momentos oportunos.

Si el juego basa su enseñanza en la aproximación de aprender haciendo, es probable que el entorno sea una simulación (metafórica o no) de un entorno real donde el jugador/estudiante pone en práctica las habilidades que se pretenden enseñar. En ese caso, la parte más importante de los contenidos didácticos no son las explicaciones teóricas o textuales, sino los propios *ejercicios* o situaciones a las que se debe enfrentar el alumno.

Cuando el entorno virtual del juego se convierte en el lugar de ejecución

de los ejercicios, las explicaciones textuales, si las hay, deberán buscar su hueco en algún sitio. En algunas aplicaciones, como Tactical Iraqi (Johnson et al., 2005), esas explicaciones están separadas por completo del entorno (en un módulo que llaman "*skill builder*"), y el jugador/estudiante debe leerlas antes de entrar en el juego (que ellos llaman "*mission game*"). En otros casos, las explicaciones pueden estar integradas dentro del propio entorno, mediante el uso de agentes pedagógicos (como en Rickel y Johnson, 1999), HUDs (como por ejemplo en CoLoBot[7]), o diálogos de personajes de una aventura gráfica (Moreno-Ger et al., 2006).

Si nos centramos en la autoría de los ejercicios integrados dentro del entorno virtual o juego, podemos decir que existen dos casos:

- Los ejercicios están definidos *en los propios mapas del juego*: cada uno de los mapas del juego es un ejercicio concreto. De esta forma, cuando dentro del ciclo de ejecución el videojuego educativo (Gómez Martín, Gómez Martín, González Calero y Jiménez Díaz, 2005b), éste se plantea qué ejercicio debe elegir, tendrá que decidir qué mapa o escenario particular ejecuta. El esquema de contenidos del juego educativo sigue siendo, por tanto, el de la figura 4.2.

- Los ejercicios están definidos independientemente del mapa o entorno en el que se ejecutan: para poder soportar esto, la arquitectura de la aplicación subyacente debe ser capaz después de establecer el nexo de unión entre las propiedades definidas en los ejercicios (estado inicial, circunstancias que deben darse en el entorno, etc.), y el mapa que se carga y que contiene las entidades del juego.

Existen numerosos ejemplos del primero de los casos, donde toda la tecnología que se utiliza gira en torno a los videojuegos. Un ejemplo claro es el videojuego para entrenamiento de bomberos de Sim Ops Studios[8], que incluye una aplicación llamada Core3D que permite crear escenarios y colocar en sitios específicos el fuego, gases y otros elementos, así como las víctimas de los accidentes y sus síntomas. En este caso el tutor humano es el que tiene que construir los ejercicios, creando mapas y colocando entidades, de tal forma que se *programa* el funcionamiento del ejercicio utilizando editores de niveles, un "*lenguaje*" específico de los videojuegos.

Nuestra propuesta metodológica se basa en la segunda de las opciones, por lo que dedicamos la sección siguiente a su exposición detallada.

[7] `http://www.ceebot.com`
[8] `http://www.simopsstudios.com`

4.4. Metodología de creación de contenidos

4.4.1. Propuesta

Cuando se desarrolla un videojuego educativo donde se presentan distintos ejercicios que el estudiante tiene que resolver, la primera opción es extrapolar el modo de crear los entornos virtuales en los videojuegos, y utilizar la misma técnica para la creación de esos ejercicios. Como veíamos en la sección anterior, eso significa que los mapas del videojuego contienen los ejercicios *cableados* en las entidades colocadas en ellos.

Sin embargo, la opción que nosotros estimamos más acertada es la *separación* de esos tipos de contenidos en dos, (i) los entornos virtuales donde se desarrolla la acción, y (ii) los ejercicios (ver figura 4.3).

Este modo de actuar, separando por un lado los datos que determinan el entorno (mapas) y por otro lado los ejercicios, soporta fácilmente las distintas relaciones que pueden darse entre ejercicios y mapas:

- El ejercicio consiste en un estado inicial del entorno, y un estado final deseado al que hay que llegar. Es posible que el entorno virtual sea el mismo durante todo el aprendizaje, por lo que sólo se requiere un mapa, mientras que los ejercicios son muchos. Un ejemplo en el área de los ITS son los ejercicios que hay que hacer dentro del sistema Steve (Rickel y Johnson, 1999, 2000) que mostrábamos en la figura 3.1, o VirPlay (Jiménez Díaz, Gómez Albarrán, Gómez Martín y González Calero, 2005a). Las acciones que se hacen dentro del entorno tienen repercusiones directas en el mismo, que, como ha empezado en un estado distinto, evolucionará de forma distinta dependiendo del ejercicio.

- El ejercicio consiste en practicar *situaciones* que se van añadiendo al entorno. El ejercicio puede partir del mismo entorno virtual (mapa) siempre, pero los sucesos que ocurren dentro de él cambian de un ejercicio a otro. Ejemplos de este tipo de aplicaciones son los simuladores de conducción, donde el estudiante siempre circula por las mismas vías, pero en ellas ocurren sucesos distintos más o menos peligrosos. El ejercicio puede especificar las distintas situaciones que deben ocurrir durante la ejecución, y las entidades del mapa, dependiendo de esos datos, se comportan de una u otra forma.

En el capítulo 6 veremos una alternativa mixta, en la que la manipulación de las entidades del entorno genera "*contenido*" nuevo dependiente del ejercicio.

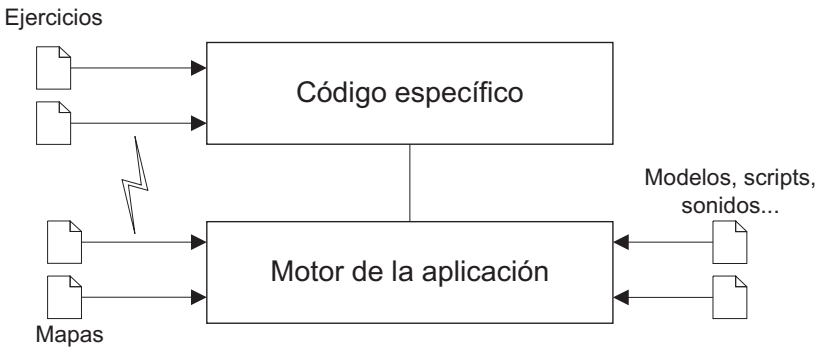


Figura 4.3: Esquema de los contenidos en juegos educativos con ejercicios y mapas separados

4.4.2. Puesta en práctica

Con nuestra metodología, existen ficheros de datos distintos para los mapas y para los ejercicios. Los primeros son leídos o interpretados, igual que antes, por el motor de la aplicación, mientras que los ejercicios son dependientes del código específico, que varía con cada dominio. Para hacer posible esta separación, los mapas están generados de tal forma que admiten la *configuración* externa de sus propiedades, dependiendo del ejercicio concreto que se quiera ejecutar. Para conseguirlo, las entidades del juego (código específico) creadas por el motor de la aplicación al cargar el mapa, utilizan los datos del ejercicio para variar su comportamiento. En la figura 4.3 se muestra el esquema general. El modo de implementar esto es tratado en la sección 5.11.

Para la creación de los ejercicios es posible que se requieran herramientas especializadas dependientes del dominio. Aunque la tendencia actual es utilizar XML como formato de intercambio de datos, y los ejercicios se prestan mucho a su uso, su edición de manera manual es cuanto menos incómoda. Por lo tanto, es recomendable disponer de herramientas especializadas que graban a estos formatos, interpretables posteriormente por la aplicación. Al tratarse de herramientas de propósito específico, pueden incluir características que hagan cómoda la creación de los ejercicios.

En cuanto a la generación de los mapas, es necesario también disponer de herramientas de modelado que permitan al menos crear la forma básica del entorno virtual (terreno y paredes), y la colocación de las entidades que definen el comportamiento del entorno y que constituyen el nexo de unión con los ejercicios.

Para la creación de estos mapas, nuestra propuesta es implementarla en dos fases:

1. En la primera fase se crea una versión inicial del mapa, utilizando

una herramienta especializada que facilite la creación de la estructura teniendo en cuenta el tipo de entorno perseguido. Por ejemplo, para un juego educativo de conducción se puede utilizar un editor especializado en carreteras, mientras que para otro tipo de juego se puede elegir uno que se base en celdas.

2. En la segunda fase se crea la versión final del mapa, donde aparecen todas las entidades que serán utilizadas para la configuración procedimental del entorno, así como el resto de entidades que intervienen en la componente lúdica. También en esta fase se puede retocar la estructura del entorno virtual.

Con esta alternativa, se aprovecha el uso de una herramienta especializada para construir de forma rápida el entorno, antes de terminar con la tarea de colocación de entidades, que es la que más tiempo consume. Además, se consigue que en las etapas tempranas del desarrollo se pueda apreciar el aspecto final del entorno, especialmente útil para el desarrollo de prototipos.

Como ejemplo de puesta en práctica de esta metodología en general, y del método de creación de mapas en particular, se puede consultar la sección 6.7, donde se describe cómo se ha realizado la generación de contenidos en JV^{2}M.

4.4.3. Ventajas

La principal ventaja de la separación física de mapas y ejercicios es la *especialización*. La arquitectura basada en datos en el desarrollo de videojuegos permite separar el papel de los programadores del de los artistas. En esta ocasión, extraer de los mapas la componente educativa permite distinguir entre el papel jugado por los diseñadores y artistas y el desempeñado por los expertos en el dominio. Esa especialización también se obtiene en los formatos de los ficheros que se utilizan. Así, los mapas pueden guardarse en un formato genérico e independiente del dominio enseñado, mientras que los ejercicios se guardan en un tipo de archivo específico.

Dado que la descripción de estos ejercicios puede ser de alto nivel, se podrán aplicar técnicas de análisis de datos sobre ellos, que ayudarán a sus autores a comprobar si se han creado suficientes o no. En concreto, en la sección 4.6.2 proponemos el uso de análisis formal de conceptos para hacerlo.

Una ventaja no menos importante con la separación es la *disminución del coste* de generación del videojuego educativo. Independientemente del tipo de relación que exista entre los ejercicios y los mapas descritas en la sección 4.4.1, lo que conseguimos con la separación es la posibilidad de utilizar el mismo conjunto pequeño de mapas en varios ejercicios distintos. De esta forma, la configuración o funcionamiento final del entorno en el que el estudiante se ve inmerso se realiza *de manera procedimental*.

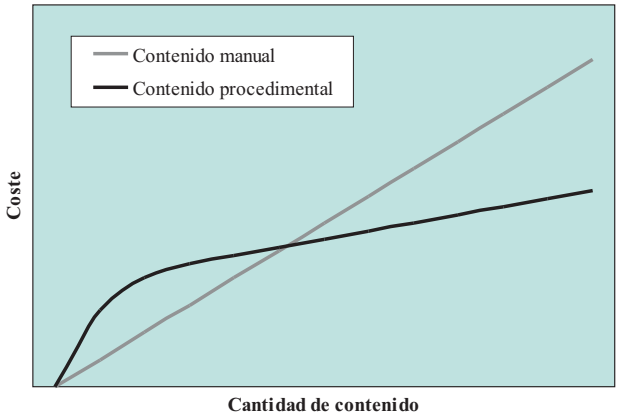


Figura 4.4: Coste de generación de contenidos de forma manual y procedimental

Esto implica una bajada en el coste de desarrollo de la aplicación, ya que es mucho más fácil la generación de ejercicios independientes del entorno de ejecución, debido a que su creación consiste únicamente en definir el "enunciado", que, en tiempo de ejecución, afectará al entorno en el que el jugador se ve inmerso. Hacer los ejercicios independientes del entorno, además, permite poder utilizarlos en distintos contextos o con distintas presentaciones.

En realidad, la disminución del coste sólo sucede cuando la cantidad de contenido de la aplicación supera un umbral. Esto es debido a que la generación o configuración procedimental de los escenarios requiere un esfuerzo inicial superior al necesario para crear unos pocos ejercicios. En la figura 4.4 se aprecia que el coste de creación de los primeros ejercicios utilizando técnicas manuales es inferior que el necesario para la generación procedimental, debido a que para esta última no vale con generar únicamente los mapas, sino que es necesario también crear los ejercicios y todo el código de construcción o configuración procedimental del entorno. Una vez pasado un umbral, el coste de ambos crece de forma lineal, pero la generación manual lo hace de forma mucho más pronunciada.

A la configuración procedimental del entorno se le puede añadir una mejora más, la creación procedimental *de los propios ejercicios*. Por ejemplo, Gómez Martín, Gómez Martín, Díaz Agudo y González Calero (2005c) presentan el uso de técnicas de adaptación de razonamiento basado en casos para generar nuevos ejercicios en JV^{2}M. En ese caso, el coste inicial (tener los primeros ejercicios funcionando) es mucho más grande, debido a la necesidad de implementar las técnicas de adaptación, pero una vez superado ese paso inicial, la función de coste no crece, ya que la generación es automática.

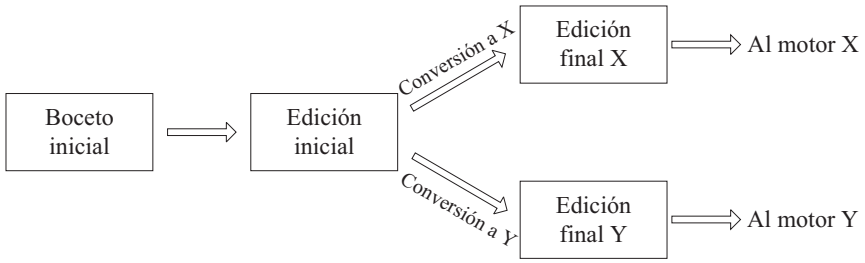


Figura 4.5: Esquema de creación de mapas

Por último, conviene mencionar una ventaja de la separación en dos fases de la generación de mapas. Como veremos en el capítulo siguiente las herramientas utilizadas en la segunda fase de creación de los mapas puede venir impuesta por el motor gráfico utilizado en la aplicación. El trabajo en dos fases permite reducir el trabajo no aprovechable cuando se decide un cambio de motor, pues la primera parte de creación del mapa sirve para ambos motores (ver figura 4.5).

4.5. Herramientas necesarias para el desarrollo

Para el desarrollo de una aplicación educativa utilizando las fases y metodología anterior, debe contarse con una serie de herramientas de soporte, algunas de ellas indispensables, y otras recomendables para maximizar el rendimiento del equipo. Este apartado hace una descripción de las más importantes (Gómez Martín, Gómez Martín y Jiménez Díaz, 2007c), comentando las distintas alternativas existentes en cuanto a implementaciones. En algunos casos, mostramos nuestra preferencia para el caso de desarrollo de videojuegos educativos de tamaño medio.

En general, podemos decir que la selección de las herramientas a utilizar se realiza en la fase de *preproducción* (ver sección 4.2.2), aunque algunas de ellas pueden incorporarse en el proceso de desarrollo en cualquier momento.

4.5.1. Compiladores, entornos integrados de desarrollo y herramientas de construcción

No cabe duda que son las herramientas indispensables en el desarrollo. En el área de la programación de videojuegos, el lenguaje estrella hoy por hoy es C++, por lo que es necesario un compilador para este lenguaje. Dependiendo de la plataforma destino, será el Microsoft Visual Studio, `gcc`, u otro compilador proporcionado por el fabricante del *hardware*.

En cuanto a los entornos integrados de desarrollo (en inglés *integrated development environment*, IDE), que permiten la edición y depuración del

código fuente, de nuevo Visual Studio es una opción posible, aunque también es utilizado comúnmente CodeWarrior[9] en la programación para algunas consolas.

En general, se puede decir que la decisión de qué compilador e IDE utilizar depende de la plataforma para la que se vaya a desarrollar el videojuego educativo. Si nos limitamos a plataforma PC con el sistema operativo Windows, la balanza se inclina irremediablemente hacia Microsoft Visual Studio .NET.

Una posibilidad, habitual en el mundo de los videojuegos, es crear aplicaciones para ser ejecutadas en varias plataformas. Para eso, habrá que abstraer las partes dependientes del *hardware*, y programar la aplicación en un lenguaje que se pueda compilar en todas.

Si el proyecto se desarrolla en varias plataformas, el proceso de compilación debe configurarse para cada una de ellas. Utilizando un lenguaje portable como C++, todos los compiladores aceptarán los mismos ficheros de código fuente como entrada. Sin embargo, en cada plataforma, el compilador utilizado será distinto y utilizará distintos parámetros para las opciones. Además, los desarrolladores utilizarán distintos IDEs para la programación del proyecto, y habrá que configurar cada uno de ellos individualmente. Esto provoca que la creación o borrado de un nuevo fichero de código fuente en el proyecto requiera reconfigurar gran cantidad de sistemas de generación del ejecutable, ya sean los propios IDEs (en ese caso, el cambio también hará que el nuevo fichero aparezca en el árbol de ficheros del IDE), o herramientas externas como `make` o `ant`.

Para solucionarlo, existen herramientas configurables con ficheros de texto independientes de la plataforma que *generan* los ficheros de configuración para los distintos sistemas de generación; por ejemplo, generarán los `.sln` y `.vcproj` para ser leídos por Visual Studio, los ficheros `Makefile` que sirven de entrada a `make`, o `build.xml` para `ant`. De esta forma, cuando se quiere añadir un nuevo fichero al proyecto, *no* se añade directamente en el entorno de desarrollo, sino en el fichero de configuración de esa nueva herramienta independiente de la plataforma. Algunas de estas herramientas pueden servir incluso para *generar* ellas mismas el ejecutable final dependiendo de la configuración establecida. Pasan entonces de ser una herramienta utilizada como paso previo a la invocación de la compilación (ya sea desde el IDE o no) a ser la propia herramienta que llama al compilador, analizando previamente las dependencias y comprobando qué ficheros deben compilarse y cuáles no. Ejemplos de estas herramientas son CMake (Martin y Hoffman, 2006), SCons[10], KJam[11] o Boost.Build[12]. Otra alternativa es desarrollar una interna

[9] `http://www.freescale.com/codewarrior/`
[10] `http://www.scons.org/`
[11] `http://www.oroboro.com/kjam/`
[12] `http://www.boost.org/tools/build/v2/index.html`

sencilla, como han hecho los desarrolladores de Nebula y Mangalore[13].

Para nuestro contexto, y gracias a que Microsoft Visual Studio .NET 2005 ha mejorado significativamente los tiempos de comprobación de dependencias, nos decantamos por utilizar el propio IDE para la tarea de generación del ejecutable. Para la creación de los ficheros que lo configuran (fichero de *solución* y ficheros de *proyecto*), utilizamos CMake (Martin y Hoffman, 2006), lo que permite fácilmente cambiar la versión del compilador, o migrar en un futuro a otras plataformas fácilmente, al evitar tener que generar los ficheros de compilación asociados.

4.5.2. Sistemas de control de versiones

Estos sistemas gestionan las distintas versiones por las que va pasando un fichero de código fuente, documento o recurso durante todo el proceso de desarrollo. La necesidad de estas herramientas está ampliamente reconocida, no sólo porque sirven como medio de copia de seguridad que permite *volver hacia atrás* ante algún fallo, sino porque permite el trabajo simultáneo de dos o más personas.

Estos sistemas de control permiten por ejemplo que dos desarrolladores modifiquen distintos (o incluso los mismos) ficheros simultáneamente, recuperar el estado anterior de ficheros o de proyectos enteros, crear ramas en el desarrollo o llevar la historia de los cambios de los ficheros.

Existen varias alternativas para el control de versiones, tanto comerciales como bajo licencia GPL o similares. El sistema por excelencia dentro del software libre es CVS (Vesperman, 2003), aunque hoy por hoy está siendo desbancado por Subversion (Collins-Sussman et al., 2004). Entre las herramientas comerciales, destacan SourceSafe de Microsoft[14], Perforce[15] y AccuRev[16].

Para el desarrollo de videojuegos educativos de tamaño medio, nos decantamos por Subversion. La poca capacidad para el trabajo colaborativo de SourceSafe y el elevado precio de Perforce y AccuRev decantan la balanza hacia las alternativas libres. Entre éstas, Subversion es superior, gracias a su gestión de actualizaciones atómicas, su capacidad de no perder la historia de un fichero al renombrarlo o moverlo de directorio, poco coste de las ramas o el concepto de cambio conjunto de varios ficheros. Además, la posibilidad de utilizar interfaz Web y la existencia de un buen cliente en Windows como TortoiseSVN hacen de Subversion una alternativa recomendable tanto para el almacenamiento de código fuente como de recursos gráficos y mapas.

[13] `http://nebuladevice.cubik.org/`
[14] `http://msdn.microsoft.com/ssafe/`
[15] `http://www.perforce.com/`
[16] `http://www.accurev.com/`

4.5.3. Sistemas de gestión del conocimiento

El conocimiento acumulado por un equipo de desarrollo es enorme. En vez de confiar en el boca a boca, es recomendable tener algún almacén centralizado que recoja toda esa información. Una manera típica de recopilarla es por medio de la creación de FAQs (del inglés, *frecuently asked questions*, o listado de preguntas frecuentes con sus respuestas), los documentos conocidos como "*how-to's*" (que contienen los procedimientos para realizar alguna tarea), tutoriales e informes técnicos.

Existen herramientas más sofisticadas, que se conocen como sistemas de gestión del conocimiento (en inglés *knowledge management systems*, KMS), que pueden ser aplicaciones hipermedia que, incluso, pueden incluir ontologías y sistemas expertos para organizar los contenidos y mejorar el soporte para búsquedas.

Para nuestro contexto creemos que este tipo de herramientas es útil para minimizar las consecuencias de que un desarrollador importante abandone la empresa. Siguiendo con las metodologías ágiles que utilizamos para el desarrollo, la opción más lógica, y por la que nos decantamos es la utilización de páginas Wiki[17] (Leuf y Cunningham, 2001). La idea básica de una página Wiki es tan simple como un sitio Web que permite a sus visitantes añadir, modificar o eliminar contenido, abriendo la puerta a la autoría de documentos de forma colaborativa.

Existen numerosas implementaciones, programadas en distintos lenguajes, y con distintas características, como TWiki[18], DokuWiki[19] o la famosa MediaWiki[20] utilizada por la Wikipedia. Para el uso que le damos como método de centralizar el conocimiento adquirido por el equipo de desarrollo, cualquiera de ellas son perfectamente válidas.

4.5.4. Sistemas de seguimiento de errores

Ya hemos visto en la sección 4.2.4 que una de las fases del desarrollo es la comprobación de que el programa no falla. Para gestionar estos errores, ya sea en esa fase del desarrollo o en otras, es preferible utilizar alguna aplicación que nos ayude.

Los sistemas de seguimiento de errores (en inglés *bug tracking systems*), son soluciones software que ayudan a los programadores y departamento de calidad (QA) a informar de los errores que han encontrado. Los responsables de programación pueden asignar distintas prioridades, y los programado-

[17] Una de las primeras implementaciones de Wikis más robustas, Wiki Web techonology, fue creada por Ward Cuningham, uno de los fundadores del eXtreme Programming, una de las metodologías ágiles más conocidas.

[18] `http://twiki.org/`

[19] `http://wiki.splitbrain.org/`

[20] `http://www.mediawiki.org/`

res particulares pueden ver la lista de errores que hay por resolver y auto asignarse la tarea de solucionar alguno. En ese caso, los sistemas permiten *bloquear* el error, para evitar que más de una persona trabaje en él. Cuando es resuelto, el programador que lo ha solventado puede *cerrar* el error, documentando incluso sus causas y solución.

Una de las ventajas no comentadas del uso de un sistema de control de versiones como Subversion y Perforce es la posibilidad de aprovechar su característica de cambio conjunto de varios ficheros. Ambos sistemas funcionan por *revisiones globales* (*changelists* en la nomenclatura de Perforce), es decir, cada actualización (o *commit*) provoca un cambio de revisión del *almacén completo* y no sólo de los ficheros implicados (como ocurre por ejemplo en CVS). Eso permite que el mensaje asociado al *commit* que soluciona un error pueda contener una referencia a ese error en el sistema de seguimiento de errores. De forma similar, el mensaje añadido en el sistema de seguimiento de errores al corregir un fallo puede hacer referencia al número de revisión concreto que solventa el fallo.

Existen muchos sistemas de rastreo de errores con distintos interfaces (vía web, e-mail o aplicaciones independientes) y licencias. Algunas compañías utilizan como herramienta interna una basada en Web montada en la red local de la empresa mientras que utilizan otras con un interfaz de aplicación independiente para la comunicación con los colaboradores externos.

Como ejemplos, podemos destacar Elementool[21] en el área del software comercial, y Bugzilla[22] y Mantis[23] entre los productos libres. El primero, muy completo, tiene un precio que para desarrollos y empresas medias puede ser excesivo. Por esa razón, la opción escogida por nosotros es o Bugzilla o Mantis; si bien el primero tiene más opciones, su complejidad de uso hace que Mantis, más sencillo de utilizar, sea a veces preferida. Mantis, por ejemplo, ha sido utilizado por Pyro Studios durante el desarrollo de varios de sus productos, como Praetorians (Arévalo, 2006).

4.5.5. Sistemas de integración continua

La integración continua (Fowler, 2006) es una de las prácticas aconsejadas por las metodologías ágiles, aunque tienen también cabida en otras metodologías de desarrollo distintas. Lo que proclama es la integración frecuente del trabajo personal de cada miembro de un equipo. De esta forma, los avances que cada programador hace son añadidos en un corto periodo de tiempo a la versión completa disponible en el almacén de control de versiones. La otra alternativa, dejar pasar días o incluso semanas antes de integrar esa nueva funcionalidad, conduce a problemas serios de integración en las

[21] `http://www.elementool.com/`
[22] `http://www.bugzilla.org/`
[23] `http://www.mantisbugtracker.com/`

últimas fases del desarrollo que alargan los plazos de entrega, incrementan el coste del desarrollo y pueden incluso llevar a la cancelación del proyecto.

Las herramientas de integración continua son aquellas que se instalan en un servidor y cuya única tarea es comprobar si la última versión disponible en el servidor de código fuente (CVS, Subversion o cualquier otro) es correcta.

Para ello, estas herramientas permiten configurarse de tal forma que cada cierto tiempo (desde pocos segundos a varios minutos u horas), comprueban si ha habido algún cambio desde la última construcción o compilación llevada a cabo. En caso de cambios, se descargan la última versión y prueban su validez. Cuando se detecta algún error, alerta, mediante distintos tipos de notificación, a los responsables del fallo, comprobando qué usuarios han modificado el código desde la última vez. La notificación puede hacerse, dependiendo de la herramienta utilizada, mediante correo electrónico, RSS, notificación mediante algún protocolo de mensajería instantánea u otros modos más extraños como la activación de sirenas.

Dependiendo de la configuración, la comprobación de validez será más o menos sencilla. Una muy habitual es realizar una mera compilación *incremental*, de forma que se comprueba que en los últimos cambios no ha habido un problema de integración debido al desarrollo en paralelo de distintos programadores, que nadie ha olvidado añadir ficheros nuevos o cualquier otro tipo de error. Otras configuraciones realizan pruebas más rigurosas, como compilaciones desde cero, para todas las plataformas destino, o ejecución de la batería de tests completa que haya definidas para el proyecto. En estos casos, dado que el tiempo de estas pruebas es mayor, la herramienta se configura para que lo haga menos frecuentemente.

Existen distintas herramientas de este tipo, aunque el propio `cron` de Linux o gestión de tareas programadas de Windows podrían valer. Entre las alternativas comerciales, destaca AntHill[24], y entre las de código abierto, CruiseControl[25].

Hay que destacar que todas estas herramientas lanzan la *compilación* de los proyectos asociados, por lo tanto, deben hacer uso tanto de compiladores como de herramientas que dirijan la invocación al compilador dependiendo de las dependencias y modificaciones en el código fuente (ver sección 4.5.1).

Nuestra experiencia nos dice que CruiseControl (ver figura 4.6) es una alternativa perfectamente válida, ya que es altamente configurable, soporta una gran cantidad de métodos de notificación y sistemas de control de versiones y dispone de interfaz web para poder comprobar el estado. Para dirigir la compilación, hace uso de `ant`, aunque ésta puede configurarse para que delegue el análisis de dependencias a otras aplicaciones externas, como `make` o el propio Visual C++.

[24] `http://www.urbancode.com/`
[25] `http://cruisecontrol.sourceforge.net/`

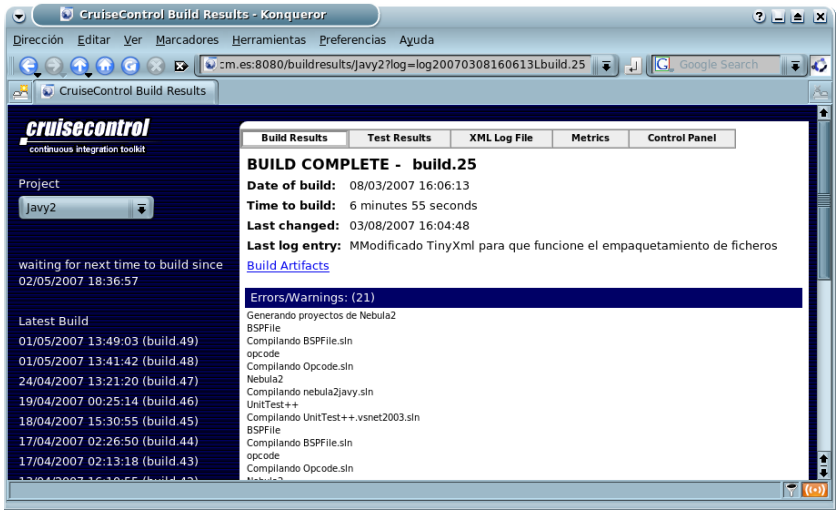


Figura 4.6: Captura de CruiseControl en el proyecto JV^{2}M 2

4.6. Otras herramientas de soporte

Ya hemos destacado en la sección 4.2.4 que durante el desarrollo de un videojuego educativo debe dedicarse una atención especial a intentar maximizar la calidad del producto, para que salga al mercado con el mínimo número de errores posible.

Sin embargo, cuando pensamos en corrección de errores, normalmente nos centramos en los que se producen *en ejecución*, y que hace que el programa se comporte de manera extraña o termine de forma inesperada. Para estos fallos, veíamos en la sección 4.5.4, existen herramientas de soporte para el seguimiento de errores, que facilitan el proceso de corrección.

Sin embargo, que una aplicación carezca de errores de ejecución (si es que eso es posible), no significa que cubra por completo las expectativas del usuario. Los creadores de aplicaciones educativas basadas en videojuegos deben maximizar la diversión del jugador y garantizar que la aplicación cubre todos los contenidos que se deseaban enseñar.

En este apartado presentamos dos técnicas que intentan ayudar en ambos aspectos, por un lado equilibrar el juego para que la dificultad percibida por los jugadores esté nivelada independientemente de su nivel de destreza y por otro lado, comprobar si los ejercicios generados para la aplicación (según lo descrito en la sección 4.3.3) son suficientes para la completa comprensión de los conceptos que se están enseñando.

Ambas técnicas utilizan lo que se conoce como Análisis Formal de Conceptos. El siguiente apartado hace una descripción de las ideas generales de este formalismo, para abordar, en las dos secciones siguientes, cada uno de

los problemas particulares que resolvemos.

4.6.1. Análisis Formal de Conceptos

El análisis formal de conceptos (en inglés *Formal Concept Analysis*, FCA), es un enfoque matemático para el análisis de datos, representación del conocimiento y gestión de la información basado en las teorías de retículos de Birkhoff (1973). Fue creado por Wille (1982), y desde entonces ha sido utilizado en muchos y muy diversos campos, como psicología, sociología, medicina, biología, lingüística o ingeniería industrial.

En esta sección nos limitaremos a hacer una descripción breve del análisis formal de conceptos. El lector interesado puede consultar literatura adicional. Wolff (1993) es un buen texto introductorio, aunque las definiciones matemáticas son mejor presentadas por Wille (1982), Ganter y Wille (1998) y Davey y Priestley (2002).

El análisis formal de conceptos se aplica a una colección de objetos, que son descritos por sus propiedades. Esas propiedades son definidas únicamente de forma binaria: un objeto puede poseer o no una propiedad o *atributo*. Por ejemplo el objeto *león*, puede venir definido por sus atributos *mamífero* y *caza*. Es importante destacar que el conjunto de datos de entrada al análisis formal de conceptos, por tanto, es *siempre* una relación binaria entre objetos y atributos que indica qué atributos tiene cada uno de los objetos. En principio, esta visión "binaria" de las propiedades de los objetos puede suponer una limitación cuando se quiere aplicar FCA a dominios complejos. Para superarla, existen técnicas que se aplican *antes* que FCA y que se encargan traducir descripciones complejas a relaciones binarias, como la discretización de los valores, o el escalado conceptual (Wolff, 1993).

Formalmente, se dice que el FCA trabaja con un *contexto formal* $\mathbb{K}$, que no es mas que una tripleta (G, M, I), donde G es el conjunto de objetos o entidades, M es el conjunto de atributos o propiedades que esos objetos pueden tener, y por último, $I \subseteq G \mathrm{x} M$ es una relación binaria, que expresa qué atributos describen cada objeto. El significado de la relación I es el siguiente: si $(g, m) \in I$, entonces el objeto g tiene la propiedad o atributo m, o, recíprocamente, el atributo m es uno de los descriptores del objeto g.

La tabla 4.1 presenta un ejemplo de contexto formal llamado "Animales"[26], en el que aparecen cinco objetos (animales), descritos por cuatro atributos. El contexto formal está expresado utilizando una tabla, en la que cada fila representa un objeto, y cada columna un atributo. Las celdas están marcadas únicamente cuando el objeto de esa fila contiene el atributo de la columna.

Si dejamos de lado por un momento FCA, podemos interpretar o definir un concepto de dos formas distintas: o bien enumerando el conjunto de

[26] Ejemplo adaptado de (Wolff, 1993).

Animales	Caza	Vuela	Plumas	Mamífero
León	×			×
Gorrión		×	×	
Águila	×	×	×	
Liebre				×
Avestruz			×	

Tabla 4.1: Ejemplo de contexto formal sobre animales

objetos que pertenecen a ese concepto (lo que se conoce como *extensión*), o enumerando el conjunto de propiedades que poseen todos los objetos de ese concepto (llamado *intensión*). En el reducido entorno descrito por el contexto formal de la tabla 4.1, podemos decir que el concepto "pájaro" tiene como extensión los objetos {Gorrión, Águila}, y como intensión {Vuela, Plumas}[27].

Lo que hemos hecho nosotros de manera intuitiva con ese concepto "pájaro" es, precisamente, lo que hace el análisis formal de conceptos de manera rigurosa: utilizar el contexto formal para extraer todos los *conceptos formales* que pueden deducirse de él. Matemáticamente, se dice que un concepto formal es una agrupación *maximal* de objetos que comparten propiedades. En el ejemplo de los pájaros, todos los objetos de la extensión tienen los atributos de la intensión, y no existe ningún otro objeto que lo haga. Sin embargo, el concepto cuya extensión es {Gorrión}, y cuya intensión es {Vuela, Plumas} *no* se considera un concepto formal para ese contexto, debido a que la extensión *no* es maximal, pues podemos añadir el objeto Águila. Debido a esta propiedad de maximalidad, existe en cierto sentido una redundancia en un concepto, ya que su extensión determina de manera unívoca su intensión y viceversa.

Formalmente, pues, para FCA, un *concepto formal* es una pareja (G', M') donde $G' \subseteq G$ es el conjunto de objetos que pertenecen a ese concepto (extensión del concepto), y $M' \subseteq M$ es el conjunto de atributos comunes (intensión). Para que sea cierto el requisito de conjuntos *maximales*, deberá cumplirse que el conjunto de atributos comunes de G' es M', y que los objetos del contexto formal que tienen los atributos M' son exactamente G'. Como veíamos anteriormente, se puede ver que el ejemplo del concepto formal "pájaro", ({Gorrión, Águila}, {Vuela, Plumas}), cumple el requisito de maximalidad, ya que los atributos comunes de los dos objetos son únicamente Vuela y Plumas, y no existen ningún otro objeto que tenga esos dos atributos. Como ejemplo adicional, existe otro concepto formal ("ave de

[27] Desde el punto de vista biológico, en realidad un águila no es un pájaro, ya que pertenece al orden de las rapaces. En esta exposición consideraremos el concepto "pájaro" como aquel comúnmente aceptado entre la población no experta: ave que vuela.

presa"), formado por ({Águila}, {Vuela, Plumas, Caza}).

El análisis formal de conceptos extrae, pues, *todos* los conceptos formales de un contexto formal como el de la tabla. Entre ese conjunto de conceptos surge de manera natural una jerarquía dirigida por la relación "superconcepto-subconcepto". En esa relación se cumple que los subconceptos tienen menos objetos (extensión más pequeña) que sus superconceptos, y, por el contrario, tienen más atributos (intensión más grande). También se cumple que la extensión de un subconcepto está incluido en la de *todos* sus superconceptos, y recíprocamente, la intensión de un superconcepto está incluida en la de sus subconceptos. Siguiendo con el ejemplo anterior, el concepto "pájaro" definido como hemos visto como ({Gorrión, Águila}, {Vuela, Plumas}), es un superconcepto de "ave de presa" o ({Águila}, {Vuela, Plumas, Caza}), donde se puede ver que se cumplen todas las propiedades anteriores.

Formalmente, la relación *superconcepto-subconcepto* define un orden parcial entre todos los conceptos formales, lo que matemáticamente constituye un *retículo*. Los retículos pueden representarse en forma de grafo donde cada uno de los nodos representa un concepto formal, y cada arista indica que los dos conceptos que une están asociados mediante la relación superconcepto-subconcepto. En la representación gráfica, la colocación de los nodos está seleccionada de tal forma que los superconceptos siempre quedan por *encima* de todos sus subconceptos.

Dado que cada uno de los nodos del retículo representa un concepto, éstos tendrán asociados su conjunto de objetos y de atributos. En la representación gráfica del retículo se puede poner, asociada a cada nodo, una etiqueta que contenga la descripción de ese concepto formal, mediante su extensión e intensión. Sin embargo, eso provocaría una representación complicada de seguir, ya que, por las propiedades vistas en el párrafo anterior, tanto los objetos como los atributos aparecerían repetidos en gran cantidad de nodos, al formar parte de la extensión (o intensión) de muchos conceptos formales. Una manera de evitarlo es etiquetar los nodos de tal forma que un atributo aparecen únicamente en el concepto más alto en la jerarquía que lo contiene en su intensión, y un objeto aparece etiquetando el nodo más bajo en la jerarquía que lo contiene en su extensión.

La figura 4.7 contiene la representación del retículo de la tabla 4.1 que estamos utilizando como ejemplo, utilizando la representación simplificada. Para averiguar la extensión u objetos de un concepto, hay que mirar los objetos que etiquetan al nodo que lo representa y los objetos de *todos sus conceptos descendientes* o subconceptos. Por su parte, los atributos de un concepto son todos aquellos que aparecen al lado del nodo en el retículo, y los atributos de sus superconceptos. Por ejemplo, el concepto en el que aparece el atributo *Caza* tiene como *extensión* el conjunto formado por los objetos León y Águila, ambos pertenecientes a sus dos subconceptos. Por su

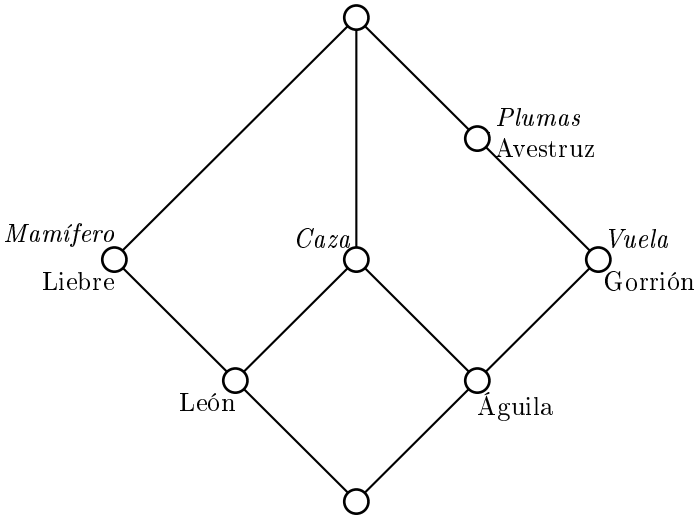


Figura 4.7: Retículo generado a partir del contexto de la tabla 4.1

parte, las propiedades o atributos del concepto en el que aparece el Águila serán *Caza*, *Vuela* y *Plumas*, ya que todos ellos etiquetan en el retículo a sus superconceptos.

Dado que la relación mostrada en el retículo es un orden parcial, se puede demostrar que éste tendrá siempre un único elemento (concepto o nodo) que es *superconcepto* del resto de nodos, y un elemento que hace de *subconcepto* de todos los demás. Estos nodos son conocidos en inglés como nodos *top* y *bottom* respectivamente. El nodo *top* representa el concepto que tiene como intensión a todos los atributos del contexto formal, mientras que el nodo *bottom* representa el concepto que tiene como extensión a todos sus objetos.

El análisis formal de conceptos ha sido utilizada con éxito en muchos ámbitos, escogiendo cuidadosamente los objetos y atributos adecuados para describir el problema, y utilizando distintas técnicas sobre los conceptos o retículos generados a partir del contexto. Por ejemplo, se ha utilizado para la ayuda en la creación de ontologías (Obitko et al., 2004), sistemas de recomendación (du Boucher-Ryan y Bridge, 2005), análisis de software (Lindig y Snelting, 1997; Eisenbarth et al., 2003; Snelting, 2005), extracción de reglas de asociación (Pasquier, 2000) o sistemas de aprendizaje máquina y clasificación (Carpineto y Romano, 1996; Kuznetsov, 2004).

Existen además numerosas aplicaciones relacionadas con FCA (Tilley, 2004), entre las que destaca ConExp[28] (ver figura 4.8), que se distribuye bajo licencia BSD. Esta herramienta es la que hemos utilizado como base

[28] `http://conexp.sourceforge.net/`

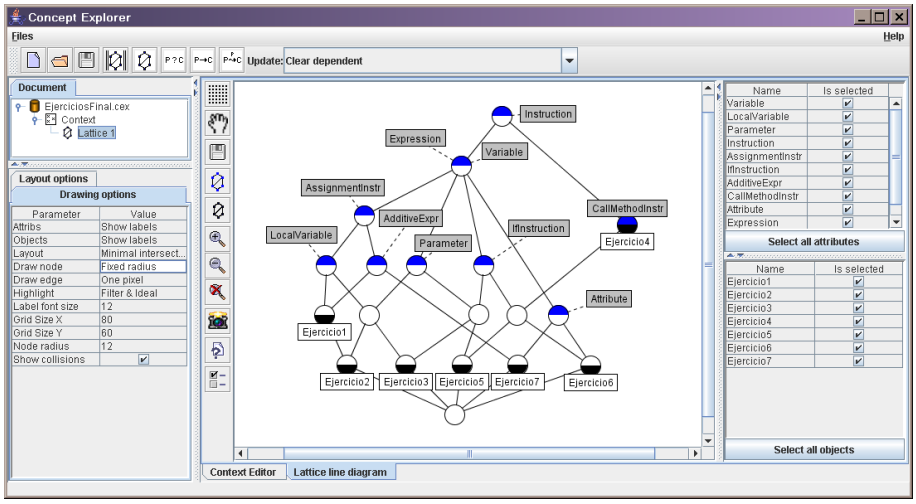


Figura 4.8: Captura de ConExp, una herramienta para FCA

para las dos secciones siguientes, que describen con detalle los dos problemas que hemos resuelto usando FCA.

4.6.2. Refinamiento de ejercicios

La metodología para la creación de contenidos expuesta en la sección 4.4 tenía como resultado la total separación de ejercicios y entornos virtuales. Gracias a esta separación, la aplicación contiene la batería de ejercicios que presentará expresados en un lenguaje independiente del género del juego.

Esto permite utilizar técnicas de análisis de datos sobre la base de ejercicios para comprobar si ésta es completa, es decir, si el estudiante, a la vista de los ejercicios, no puede deducir de ellos ninguna conclusión incorrecta.

El análisis del problema de comprobación de si la base de ejercicios es completa o no nos llevó a presentar en Díaz Agudo, Gómez Martín, Gómez Martín y González Calero (2005) una solución general basada en el análisis formal de conceptos. La generalización consiste en extrapolar los resultados obtenidos sobre la base de ejercicios de una aplicación educativa a un sistema de razonamiento basado en casos, en la que lo que en el sistema educativo eran ejercicios, se convierten en casos, y el problema de comprobar si la base de ejercicios es completa se convierte en comprobar la *cobertura* de la base de casos. Así, como veremos a continuación, el análisis formal de conceptos se convierte en una técnica que ayuda al mantenimiento de la base de casos, para refinar el conocimiento contenido en ella de tal forma que maximicemos los resultados obtenidos en la fase de recuperación del caso. En particular, FCA sirve para encontrar *huecos* en la cobertura dada por la base de casos, es decir, casos que no están disponibles en la base de casos pero que pueden

Figura 4.9: Vista parcial de la ontología de elementos de compilación de Java

ser necesitados en los procesos de razonamiento.

La técnica utilizada consiste en aplicar el *aprendizaje conceptual* disponible en el análisis formal de conceptos (Wille, 1989, 1992; Stumme, 1999). Este tipo de aprendizaje es en realidad un procedimiento de adquisición de conocimiento por refinamiento sucesivo. Haciendo un análisis de los conceptos creados por el FCA a partir de un contexto formal, se pueden inferir determinadas conclusiones o implicaciones que pueden o no ser ciertas. Analizando esas implicaciones se puede determinar que el contexto formal debe ser extendido con otros objetos o atributos. Por ejemplo, en el reducido contexto formal de la tabla 4.1, el análisis formal de conceptos detecta que todo animal que vuela tiene plumas. El aprendizaje conceptual consiste precisamente en detectar esa implicación y refutarla, añadiendo al contexto formal (por ejemplo) la abeja, que vuela pero no tiene plumas.

Volviendo al contexto de las aplicaciones educativas, veamos cómo utilizamos esto en nuestra aplicación, JV^{2}M presentada en el capítulo 6, desarrollada utilizando la metodología de creación de contenidos descrita.

El sistema, relacionado con la enseñanza de la compilación de lenguajes orientados a objetos, maneja como ejercicios programas en Java. El módulo pedagógico tiene, entre otras cosas, una ontología de conceptos relacionados con el lenguaje, que puede verse parcialmente en la figura 4.9. El concepto "raíz" es el elemento de compilación, cuyos subconceptos son las instrucciones (`while`, `if`, etc.), las expresiones (sumas, multiplicaciones y operaciones lógicas entre otras) y las variables (ya sean variables locales, parámetros de

```
public void f() {
    int a;
    a = 3 + 5;
}
```

(a) Ejercicio 1

```
public void f(int b) {
    int a;
    a = 3 + b;
}
```

(b) Ejercicio 2

```
public void f(int b) {
    int a;
    if (b == 0)
        a = 4;
}
```

(c) Ejercicio 3

Figura 4.10: Base de ejercicios inicial antes del refinamiento con FCA

métodos o atributos). Cada ejercicio del sistema está relacionado con uno o más conceptos de esta ontología. Cuando el módulo pedagógico (ver sección 3.2.3) decide los conceptos que quiere que el usuario practique, consulta la base de ejercicios y selecciona el más adecuado de acuerdo a la estrategia pedagógica seguida.

Para utilizar el aprendizaje conceptual, expresamos los ejercicios de la base de ejercicios utilizando un contexto formal. Para ello, consideramos cada uno de los ejercicios como un *objeto*, y cada concepto de la ontología como un *atributo*, de tal forma que un objeto tendrá asociado un atributo cuando ponga en práctica o enseñe el concepto asociado a él.

Para ilustrar el proceso de refinamiento de la base de casos, lo ejemplificaremos utilizando como base de ejercicios la expuesta en Díaz Agudo et al. (2005). Supongamos que nuestro experto en el dominio ha ideado únicamente los tres ejercicios Java para el sistema educativo que aparecen en la figura 4.10.

El ejercicio 1 (figura 4.10a), por ejemplo, al utilizar una variable local `a`, pone en práctica el concepto "LocalVariable", y por lo tanto también su concepto padre, "Variable". Dado que tiene una instrucción de asignación, también está relacionado con "Instruction" y "AssignmentInstruction". Por último, el código `3+5` es responsable de que el ejercicio también enseñe los conceptos "Expression" y "AdditiveExpresion"[29].

La diferencia entre el ejercicio 1 y el ejercicio 2 (figura 4.10b), es que este último tiene además un parámetro, por lo que también enseña el concepto "Parameter".

Por último, en el ejercicio 3 (figura 4.10c) la expresión aritmética ha sido

[29]Durante todo el ejemplo obviaremos el concepto "elemento de compilación" de la ontología, por ser padre de todos los demás.

Ejercicios	Variable	LocalVariable	Parameter	Instruction	AssignmentInstr	IfInstruction	Expression	AdditiveExpr	BooleanExpr
Ejercicio 1	×	×		×	×		×	×	
Ejercicio 2	×	×	×	×	×		×	×	
Ejercicio 3	×	×	×	×	×	×	×		×

Tabla 4.2: Contexto formal de la base de ejecicios inicial

sustituida por una instrucción `if` con una expresión booleana, lo que hace que se enseñen los conceptos "IfInstruction" y "BooleanExpression", pero no se ponga en práctica "AdditiveExpression".

El análisis de los tres ejercicios conduce a la creación del contexto formal que tiene como objetos los tres ejercicios, y como propiedades la lista de todos los conceptos de la ontología de Java mostrada en la figura 4.9. En la tabla 4.2 mostramos el contexto formal resultante, eliminando aquellos conceptos que no son puestos en práctica por ninguno de los ejercicios.

Al aplicar el análisis formal de conceptos a FCA, obtenemos seis conceptos formales cuya representación en forma de retículo aparece en la figura 4.11. Cabe destacar de este retículo que el concepto *top*, que agrupa todos los objetos del contexto formal tiene, al contrario de lo que es habitual, *cinco atributos*, debido a que todos los ejercicios ponen en práctica los conceptos asociados a ellos.

Si obtenemos las implicaciones válidas sobre los conceptos formales extraídos para comenzar con el aprendizaje conceptual, la primera implicación que se obtiene es, precisamente, que todos los ejercicios enseñan los conceptos "Variable", "LocalVariable", "Instruction", "AssignmentInstr" y "Expression" (es decir, aparece la regla $\{ \emptyset \rightarrow$ *Variable*, *LocalVariable*, *Instruction*, *AssignmentInstr*, *Expression*$\}$). En el momento de refinar la base de ejercicios, el tutor o experto humano, guiado por este aprendizaje conceptual, introduce un nuevo ejercicio o ejemplo donde esto no sea cierto. El resultado es un nuevo ejercicio, que aparece en la figura 4.12a, que únicamente enseña los conceptos "Instruction" y "CallMethodInstruction".

Una nueva iteración del aprendizaje conceptual nos dice que en el contexto formal o base de ejercicios que tenemos, siempre que aparece una expresión booleana, también tenemos una variable, una variable local, un parámetro, una expresión y una instrucción de asignación y condicional (*BooleanExpression* $\rightarrow$ *Variable*, *LocalVariable*, *Parameter*, *Expression*, *AssinmentInstr*, *IfInstruction*). Para romper la necesidad de la existencia de una variable lo-

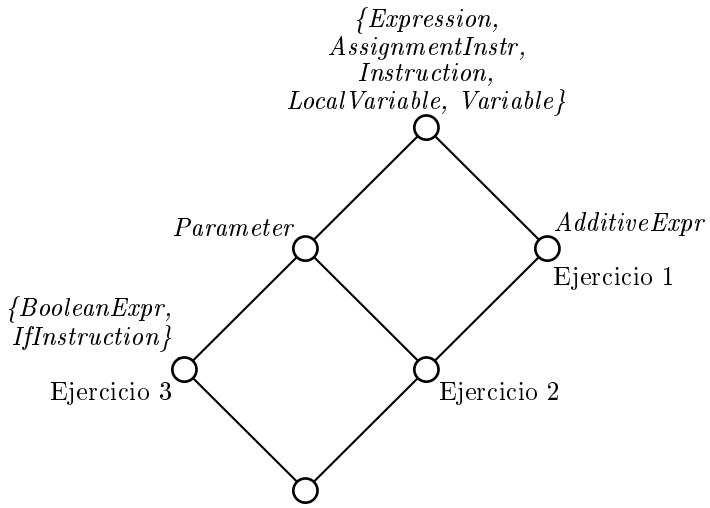


Figura 4.11: Retículo de la base de ejercicios inicial

cal cuando tenemos una expresión booleana, el experto introduce en la base de ejercicios uno en el que la expresión booleana se construye utilizando el valor de un parámetro, como en el ejercicio de la figura 4.12b.

Con este nuevo ejercicio, extendemos el contexto formal con un quinto objeto, que tiene como atributos la lista de los conceptos que pone en práctica: "Instruction", "CallMethodInstruction", "IfInstruction", "Expression" y "BooleanExpression".

Al calcular los conceptos formales y el nuevo retículo, se observan, entre otras, las siguientes implicaciones: {*BooleanExpr* → *Variable*; *BooleanExpr* → *Parameter*; *BooleanExpr* → *LocalVariable*; *BooleanExpr* → *Instruction*; *BooleanExpr* → *AssignmentInstr*; *BooleanExpr* → *IfInstruction*; *BooleanExpr* → *Expression*}.

El experto analiza la primera de ellas ({*BooleanExpr* → *Variable*}), y la da por buena dentro del dominio (pues no es habitual encontrar expresiones booleanas que no involucren variables, como `3<6`). Al comprobar la segunda implicación ({*BooleanExpr* → *Parameter*}) se comprueba que ésta no tiene por qué cumplirse en el dominio. En nuestra reducida base de casos se cumple, debido a que en todos los ejercicios donde hemos utilizado un parámetro, hemos hecho uso de una expresión booleana. Para evitarlo, se añade un ejercicio en el que la expresión booleana utiliza un atributo del objeto en vez de un parámetro (ver figura 4.12c).

Este nuevo ejercicio hace aparecer un nuevo concepto en el contexto formal que hasta este momento no habíamos enseñado, el concepto "Attribute". La consecuencia inmediata es que FCA infiere que éste aparece siempre relacionado con el concepto "IfInstruction", ya que siempre que aparece el

```
public void auxf() {
   System.out.
       println("En auxf()");
}
public void f() {
   auxf();
}
```

(a) Ejercicio 4

```
public void f(int a) {
   if (a > 0)
      System.out.
          print("Positivo");
}
```

(b) Ejercicio 5

```
public void f() {
   if (this.age > 18)
      System.out.
          print("mayor");
}
```

(c) Ejercicio 6

```
public void cumple() {
   edad = edad + 1;
}
```

(d) Ejercicio 7

Figura 4.12: Ejercicios añadidos mediante refinamiento con FCA

primero también lo hace el segundo. Para romper esta dependencia, el experto introduce un nuevo ejercicio (en la figura 4.12d), para indicar que los atributos pueden aparecer sin la instrucción `if`.

Al ejecutar de nuevo FCA con el ejercicio recién añadido, se detecta que siempre que aparece el concepto "Attribute" lo hacen también los de "Variable", "Instruction" y "Expression". El experto comprueba que esto es cierto en el dominio, ya que:

- Según la ontología de Java que utilizamos (figura 4.9), el concepto "Attribute" es un subconcepto de "Variable", por lo tanto el segundo aparecerá siempre que lo haga el primero.
- Todos los ejercicios deben tener al menos una instrucción Java.
- Los accesos a atributos se hacen siempre a través de expresiones.

Aunque aún quedan implicaciones que no son válidas en el dominio y que deben ser refutadas por un experto añadiendo nuevos ejercicios (casos), pararemos aquí, dando una idea de cómo seguiría el proceso. La nueva implicación no válida detectada por el aprendizaje incremental es $\{BooleanExpr \rightarrow IfInstruction\}$, ya que las expresiones booleanas aparecen siempre de la mano de `if`'s. El experto, por tanto, deberá romper la relación introduciendo una nueva instrucción, como el `while`, que requiere expresiones booleanas pero no utiliza la expresión `if`. Eso provocará que la vieja e incorrecta dependencia se convierta en dos dependencias nuevas, $\{IfInstruction \rightarrow BooleanExpr\}$ y $\{WhileInstruction \rightarrow BooleanExpr\}$, ambas correctas.

Ejercicios	Variable	LocalVariable	Parameter	Instruction	AssignmentInstr	IfInstruction	Expression	AdditiveExpr	BooleanExpr	CallMethodInstr	Attribute
Ejercicio 1	×	×		×	×		×	×			
Ejercicio 2	×	×	×	×	×		×	×			
Ejercicio 3	×	×	×	×	×	×	×		×		
Ejercicio 4				×						×	
Ejercicio 5	×		×	×		×	×		×	×	
Ejercicio 6	×			×		×	×		×	×	×
Ejercicio 7	×			×	×		×	×			×

Tabla 4.3: Contexto formal de la base de ejercicios final

El contexto formal resultante del proceso de añadir los cuatro nuevos ejercicios mostrados en la figura 4.12 aparece en la tabla 4.3.

El retículo final asociado al contexto formal de la tabla 4.3 es el que aparece en la figura 4.8 de la página 110 que mostraba la herramienta ConExp.

El proceso de refinamiento de la base de ejercicios utilizando FCA puede, como hemos dicho, extrapolarse a bases de casos generales donde un experto quiere comprobar si la cobertura que ésta proporciona es lo suficientemente extensa. En vista del ejemplo, las reglas o implicaciones que se generan utilizando FCA y que son las que dirigen el proceso de aprendizaje conceptual pueden interpretarse como ciertas, en cuyo caso no se requiere ninguna acción, o falsas, lo que provocará que el experto añada un nuevo caso a la base de casos.

Para concluir, en nuestro ejemplo, las reglas pueden ser ciertas por dos razones: por ser el resultado de dependencias entre atributos debido a la jerarquía de conceptos subyacente (como en nuestro caso la regla $\{$*LocalVariable* $\rightarrow$ *Variable*$\}$), o por ser ciertas en el dominio, como por ejemplo $\{$*IfInstruction* $\rightarrow$ *BooleanExpr*$\}$. En ese último caso, las reglas deducidas pueden ser utilizadas por el módulo pedagógico para dirigir el orden en el que se presentan los conceptos al alumnos.

4.6.3. Nivelado del juego

Como decíamos en la sección 4.2.4, el esfuerzo que necesita realizar un jugador para superar los niveles y oponentes tiene un efecto importante en el entretenimiento percibido. Los jugadores ocasionales prefieren enemigos vencibles, mientras que los habituales prefieren enemigos que supongan un

gran reto.

Para conseguir que todos los tipos de usuarios estén conformes con la dificultad del juego, lo habitual es que al principio de una partida, el jugador pueda indicar a qué *nivel de dificultad* desea enfrentarse[30]. El diseñador del juego, por tanto, decide una serie de parámetros que cambiarán entre un nivel de dificultad y otro. Esta decisión, sin embargo, no es fácil. Si bien es cierto que en juegos sencillos los parámetros que hay que cambiar son evidentes, no lo resultan tanto en otros. Por ejemplo, parece claro que en el juego del buscaminas el incremento del tamaño del tablero o del número de minas afecta a la complejidad. Sin embargo, en juegos no triviales, el enlace o dependencia entre los parámetros del juego y su complejidad no está tan definida.

Para ayudar precisamente en el nivelado del juego, presentamos en Gómez Martín, Gómez Martín, González Calero y Díaz Agudo (2006b) un uso del análisis formal de conceptos que puede ayudar en la tarea. Para explicar el proceso, el primer paso es entender cuál es el trabajo del diseñador en cuanto a *nivelado* de la dificultad del juego.

Al crear un juego cuyo nivel de dificultad es estático y elegido previamente por el jugador, el diseñador debe, como primer paso, decidir entre cuántos niveles de dificultad va a poder elegir el jugador. Para eso, consideraremos L al conjunto de esos niveles, por ejemplo:

$$L = \{facil, normal, dificil\}$$

Por otro lado, el diseñador tiene a su disposición distintos parámetros que puede cambiar para establecer la dificultad del juego. Por ejemplo, en un juego FPS, puede cambiar el número de enemigos en el mapa, o la precisión del arma. Llamaremos al conjunto de esos parámetros de diseño P. Todos los elementos $p_i \in P$ tienen un dominio específico, que denotaremos con $D(p_i)$. Por ejemplo, el dominio del parámetro que indica el número de enemigos del mapa son los números naturales.

$$P = \{numeroenemigos, precisionarma\}$$

$$D(\text{numero enemigos}) = \mathbb{N}^+$$

La tarea del diseñador en el momento de nivelar el juego, por tanto, es decidir, para cada uno de los elementos $l_i \in L$, el valor de *todos* los parámetros $p_i \in P$, es decir, tiene que definir las funciones:

$$t_i : L \rightarrow D(p_i)$$

[30] Existen investigaciones en marcha para añadir *dificultad adaptativa* en los juegos (Spronck et al., 2006; Charles et al., 2005; Ponsen y Spronck, 2004), aunque hoy por hoy los que lo incorporan son minoritarios.

o, agrupándolas todas en una única función t (por representar la *tarea* del diseñador):

$$t : L \rightarrow D(p_1) \times \ldots \times D(p_n)$$

que es la función que recibe el nivel de dificultad, y proporciona la tupla con todos los valores de cada parámetro.

Como ya hemos comentado, cuando el juego es sencillo, estos valores se deciden *ad hoc*, utilizando intuición y sentido común. Sin embargo, cuando los juegos son más complejos, se necesitan otros mecanismos.

Uno de estos mecanismos es realizar pruebas con grupos de jugadores reales en etapas tempranas del desarrollo. El conjunto de personas (al que llamaremos T) se elige de tal forma que el nivel de habilidad esté distribuido de forma uniforme, aunque son enfrentados todos a la misma *versión* del juego, independientemente de su nivel estimado. Durante las partidas, los diseñadores observan la forma en la que cada uno se desenvuelve, para ayudarle a entender *cómo la gente juega a su juego*. La aplicación, además, suele ser una versión modificada que deja registrado en ficheros externos información valiosa que posteriormente los diseñadores podrán analizar.

Nuestra técnica se basa en la premisa de que somos capaces de *estimar* la habilidad de los jugadores, teniendo en cuenta su nivel con juegos similares, su afición a los videojuegos, etc. Es decir, contamos con una función que nos indica el nivel de dificultad de cada jugador:

$$s : T \rightarrow L$$

Como hemos dicho, la versión modificada del juego genera un registro que después podremos analizar. Formalmente, podemos establecer que esa versión especial es capaz de *medir* un conjunto de parámetros, que de aquí en adelante llamaremos M, como el tiempo que tarda el jugador en pasar una fase o el número medio de disparos antes de acertar a un enemigo. Igual que antes, cada uno de los valores $m_i \in M$ tendrá un dominio determinado, $D(m_i)$.

Inspeccionando los registros generados, podremos averiguar, para cada jugador, los valores de los parámetros medibles de M, es decir, tendremos definida la función:

$$f_i : T \rightarrow D(m_i)$$

o, agrupándola en una función general que devuelve una tupla:

$$f : T \rightarrow D(m_1) \times \ldots \times D(m_n)$$

Los datos extraídos de f deben servir a los diseñadores para extraer conclusiones, como por ejemplo que los jugadores noveles de un FPS (por

ejemplo, el conjunto $\{j|j \in T, s(j) = facil\}$), necesitan mucha más munición que los expertos. Con esas conclusiones, el diseñador decidirá hacer que en los niveles fáciles sea sencillo conseguir munición extra.

Para que las conclusiones a las que se llegan a partir de la información generada pueda aplicarse, por tanto, es necesario que exista una dependencia entre los valores que medimos en el experimento (M), y el conjunto de parámetros del juego que podemos cambiar (P). En otras palabras, asumimos que los datos obtenidos de un jugador j, $f(j)$, serían distintos si los hubiéramos medido con valores distintos de los parámetros P[31].

Desgraciadamente, la cantidad de información recopilada en un experimento como el descrito puede ser demasiado grande para permitir a los diseñadores un análisis manual. Nuestra aportación consiste en aplicar análisis formal de conceptos para procesarla y proporcionar pistas que ayuden en el equilibrado del juego.

En esta ocasión, utilizamos la capacidad de FCA de extraer *reglas de dependencia* o asociación (Hipp et al., 2000). Una regla de asociación es una expresión con la forma $A \rightarrow B$ donde tanto A como B son atributos, y que puede interpretarse como que todos los objetos del contexto formal que contienen los atributos en A contienen también los de B.

Las reglas de asociación se caracterizan por dos parámetros:

- Confianza: expresa la *probabilidad* de que la regla se cumpla, o en otras palabras, el porcentaje de objetos que, teniendo todos los atributos de A, tienen también los de B.

- Soporte: indica cuántos objetos cumplen la regla, es decir, el número de objetos que entre sus propiedades tienen todas las de los dos conjuntos, A y B. El valor puede darse tanto en manera absoluta (número de objetos), como en porcentaje (número de objetos entre número total de objetos).

En la tabla 4.4 se muestran como ejemplo las reglas de dependencia extraídas del contexto formal presentado en la tabla 4.1 que tienen una confianza de más del 50 % y un soporte de al menos un objeto[32].

Los algoritmos de extracción de reglas utilizando análisis formal de conceptos son capaces de obtener de forma eficiente todas las reglas por encima de un determinado nivel de confianza y soporte. Aunque existen distintos algoritmos, en nuestro caso hemos utilizado el expuesto por Duquenne y Guigues (1986) para extraer las reglas exactas (con un cien por cien de

[31] Como ejemplos de valores medibles M que no cumplen esto, tendríamos el número de veces que un jugador pausa el juego, o el número de saltos que da.

[32] La última regla de la tabla no tiene atributos en la parte izquierda; lo que indica es que todos los objetos tienen Plumas con un 60 % de confianza.

Regla	Confianza	Soporte
Caza Plumas → Vuela	100 %	1
Vuela → Plumas	100 %	2
Plumas → Vuela	66'6 %	2
→ Plumas	60 %	3

Tabla 4.4: Reglas de dependencia del contexto formal de la tabla 4.1

confianza), y el de Luxenburguer (1991) para las no exactas. Para la implementación nos hemos basado en la disponible en ConExp.

El método de uso consiste en generar, para cada uno de los jugadores en T un *objeto* en el contexto formal que posteriormente procesaremos. Los atributos de ese contexto son de dos tipos:

- Nivel de juego: existe un atributo por cada elemento $l_i \in L$. Cada objeto tendrá únicamente la propiedad que marca el nivel en el que está catalogado según la función s.
- Valores de los parámetros medibles: cada uno de los parámetros que se miden en el experimento ($m_i \in M$) son traducidos a uno o más atributos en el contexto formal. Dado que normalmente esos parámetros son numéricos, es probable que se requiera hacer escalado de sus valores (Wolff, 1993), como veremos en el ejemplo que mostraremos a continuación.

Las reglas de dependencia extraídas con FCA relacionan unos atributos con otros. Algunas de las reglas serán el resultado de dependencias dadas en el dominio en cuestión, como por ejemplo las resultantes del escalado de atributos, o las definidas explícitamente en el juego (por ejemplo, la que relaciona el número de balas gastadas con el número de munición recogida). Las reglas que interesan para ayudar al diseñador son todas aquellas que tienen en su parte izquierda atributos relacionados con el *nivel* del juego (el primer grupo de atributos que mencionábamos):

$$l_i \rightarrow b_1 \ ... \ b_n \qquad \text{(confianza del c \%)}$$

donde l_i representa el atributo correspondiente a ese nivel de dificultad, y $b_1 ... b_n$ son los parámetros medibles codificados para poder expresarlos dentro del contexto formal.

Como prueba de concepto, hemos ayudado a un supuesto diseñador del Tetris a nivelar el juego (Gómez Martín, Gómez Martín, González Calero y Díaz Agudo, 2006b). El experimento comienza con la modificación de un clon del tetris llamado U61 (Mauduit, 2003). La aplicación, que se muestra en la figura 4.13, está disponible bajo licencia GPL, lo que nos permitió

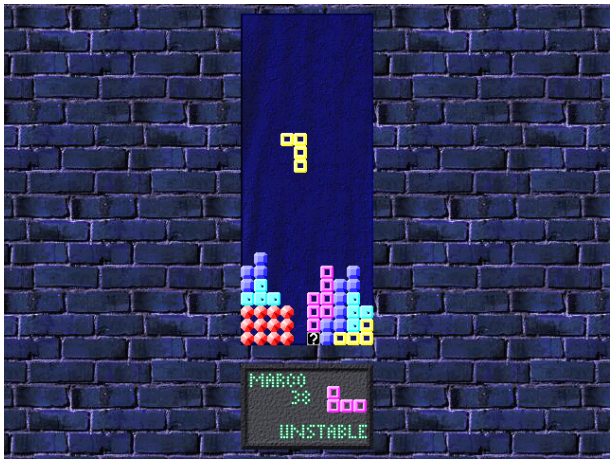


Figura 4.13: Captura de U61, clon del tetris

modificarla para generar ficheros con información sobre las partidas que se juegan.

El conjunto de parámetros medibles (M) que se guardan (además del nombre del jugador) es:

- La puntuación conseguida.
- El número de piezas colocadas.
- Para cada uno de los siete tipos de piezas que hay:
  - El tiempo medio en colocar cada ficha.
  - El número de cambios de dirección medios. Cuando una pieza está cayendo, el jugador puede elegir moverla en una u otra dirección. Si el usuario, una vez empezada a mover la ficha, cambia de opinión y la empieza a mover en sentido contrario, queda registrado.
  - El número medio de veces que una ficha golpea con los muros laterales o con otras fichas ya colocadas previamente. Existen dos valores, uno para los choques al mover la ficha hacia la izquierda, y otro hacia la derecha.
  - El número medio de veces que una ficha golpea muros u otras fichas cuando se rota en cada una de las dos direcciones posibles.
  - El número medio de espacios dejados por la ficha al colocarse en el tablero. Los huecos se miden intentando colocar la ficha una posición más abajo de la posición final. Se registra un *hueco* por cada cuadrado del tetraminó que no colisiona con alguna ficha previa.

Partidas	Fácil	Medio	Difícil	$<_{300}$	$<_{400}$	$<_{500}$	$<_{600}$	$<_{700}$	$<_{800}$	$<_{900}$	$\geq_{900}$
Jugador 1			×	×	×	×	×	×	×	×	
Jugador 2	×					×	×	×	×	×	
Jugador 3		×						×	×	×	
Jugador 4	×										×

Tabla 4.5: Contexto formal parcial de las partidas de tetris

Para el estudio, la información de cada tipo de ficha fue agrupada, de tal forma que para el análisis con FCA generamos un contexto formal que únicamente tenía las medias *totales*, en vez de su distribución por tipo de ficha.

Para poder crear el contexto, hicimos el *escalado* de los parámetros medibles. Por ejemplo, para el parámetro que indica el tiempo medio de colocación de cada ficha, creamos 8 atributos distintos, para definir los intervalos desde los 300 milisegundos hasta los 900, añadiendo también la posibilidad de que el valor fuera menor de 300 o mayor de 900.

La tabla 4.5 muestra parcialmente el contexto formal generado en el experimento. Como vemos, existe una fila/objeto por cada uno de los jugadores (partidas). Existen después tres atributos, que se corresponden con los distintos niveles de dificultad ($l_i \in L$), que en el experimento establecimos como {facil, medio, dificil}. En la tabla se ven también los atributos para el parámetro que define el tiempo medio de colocación por ficha. Como se puede ver, el primer jugador tiene una media inferior a los 300 milisegundos, el segundo está entre los 400 y los 500 milisegundos, mientras que el cuarto tarda más de 900 milisegundos de media.

Al extraer del contexto formal las reglas de dependencia ocultas en el contexto formal, surgen, como dijimos, una gran cantidad de reglas exactas debido al mecanismo de escalado, como:

$$<_7 \quad \rightarrow \quad <_8,\ <_9 \quad \text{(conf. 100 \%)}$$

Ignorando esas reglas debidas al escalado, el análisis formal de conceptos extrae otras que sí son útiles al diseñador, como:

$$\begin{aligned}
facil &\rightarrow <_{800} && \text{(conf. 85.71 \%)}\\
medio &\rightarrow <_{800} && \text{(conf. 100 \%)}\\
dificil &\rightarrow <_{500} && \text{(conf. 100 \%)}\\
dificil &\leftrightarrow <_{400} && \text{(conf. 75 \%)}\\
<_{300} &\rightarrow dificil && \text{(conf. 100 \%)}\\
\geq_{900} &\rightarrow facil && \text{(conf. 100 \%)}
\end{aligned}$$

Las conclusiones a las que llega el diseñador son las que, en este caso,

habríamos podido extraer utilizando sentido común: cambiar la velocidad a la que caen las fichas (un parámetro de diseño en P), afecta directamente en la experiencia del jugador. Eso es debido a que ese parámetro está directamente relacionado con el parámetro medible (del conjunto M) que indica el tiempo que tarda el usuario en poner una ficha, que, según las reglas anteriores, está relacionado con el nivel del jugador.

Aunque el ejemplo anterior llega a unos resultados más que evidentes, debemos destacar otros resultados menos intuitivos a los que podemos llegar al analizar el resto de reglas extraídas (que no listamos). De ellas se puede concluir que los usuarios avanzados golpean con más frecuencia las fichas que están colocando con las paredes y fichas ya fijas en el tablero[33], por lo que una posible decisión de diseño podría haber sido hacer que cada vez que se produjera uno de estos choques se penalizara bajando la ficha una posición. Según las conclusiones, este cambio afectaría a los jugadores avanzados, dejando la experiencia de juego prácticamente intacta para los noveles.

Una última conclusión a la que se llega en el análisis de los datos es que el número de cambios de dirección medio realizan los jugadores (es decir, el número de veces que "dudan") no está directamente relacionado con su habilidad, por lo que el diseñador no debe plantearse modificar ningún aspecto del diseño relacionado con este parámetro medido.

Resumen

En este capítulo hemos detallado el proceso de desarrollo de un videojuego educativo. Hemos visto (sección 4.2) que éste puede dividirse en cinco fases de forma similar a las comúnmente aceptadas en la creación de un videojuego con fines lúdicos. En esa misma sección, describíamos los objetivos y tareas que deben realizarse en cada una de estas fases, en comparación con lo necesario para la implementación de una aplicación lúdica sin contenido pedagógico.

Una de las diferencias más importantes es la *creación del contenido* o datos que la aplicación utilizará para presentar al usuario. Después de describir brevemente distintas alternativas para su creación en la sección 4.3, hemos propuesto una metodología para su creación en la sección 4.4.

Después hemos pasado a analizar y recomendar las herramientas de soporte más adecuadas para el desarrollo de videojuegos educativos (sección 4.5).

Por último, en la sección 4.6 hemos visto cómo podemos hacer uso del análisis formal de conceptos como una herramienta de análisis auxiliar para ayudar en dos aspectos importantes que se deben comprobar del funcionamiento de una aplicación educativa: el nivelado del nivel de dificultad del

[33] Suponemos que esto es debido al uso de la repetición de teclado.

juego, y la cobertura de los ejercicios disponibles en la aplicación.

En el próximo capítulo describimos la propuesta arquitectónica para el desarrollo de sistemas educativos basados en videojuegos. Esta arquitectura tiene como *premisa* que el proceso de generación de contenidos para las aplicaciones que se rigen por ella han sido creados utilizando la metodología expuesta en este capítulo.

Notas bibliográficas

Para una descripción breve del proceso de desarrollo de un videojuego, se puede consultar Llopis (2005a), aunque si se desean detalles sobre la parte de diseño y nivelado del juego, definitivamente las guías son Crawford (1984) (cuyo capítulo 5 es muy enriquecedor, para comprobar el avance que ha sufrido el área), Rollings y Adams (2003) y Crawford (2003) (este último una revisión del publicado en 1984). Un análisis de las herramientas necesarias aparece en Gómez Martín, Gómez Martín y Jiménez Díaz (2007c).

Para la creación de contenidos en videojuegos son ilustrativos los documentos (no académicos) que sirven de guía o manual de los distintos editores de mapas liberados por distintas compañías, como el del editor SandBox utilizado para Far Cry (Crytek, 2004). Para comprender hasta qué punto los editores de niveles *programan* el documento de diseño utilizando entidades, es interesante Martel (2006). Para terminar con la literatura relacionada con la creación de contenidos para aplicaciones educativas, la explicación de Gómez Martín, Gómez Martín, Díaz Agudo y González Calero (2005c) resulta de especial relevancia, ya que muestra una forma de generar automáticamente ejercicios, que luego pueden servir de entrada para la creación de los entornos virtuales.

Relacionado con el análisis formal de conceptos, una bibliografía on-line bastante extensa puede encontrarse en `http://www.fcahome.org.uk`. Existen también algunos libros monográficos dedicados únicamente a FCA, como Ganter y Wille (1998) y Davey y Priestley (2002). Por último, destacar la existencia de dos conferencias internacionales dedicadas: la serie de conferencias internacionales en análisis formal de conceptos (ICFCA) iniciada en 2003, y la conferencia internacional en estructuras conceptuales (ICCS), desde 1995. Con respecto al uso que damos nosotros al FCA, ya hemos dicho durante su exposición que el análisis de los ejercicios o contenido pedagógico ha sido presentado en Díaz Agudo, Gómez Martín, Gómez Martín y González Calero (2005), mientras que el de nivelado de juego aparece en Gómez Martín, Gómez Martín, González Calero y Díaz Agudo (2006b).

Capítulo 5

Arquitectura para el desarrollo de sistemas educativos basados en videojuegos

Todo método consiste en el orden y disposición de aquellas cosas hacia las cuales es preciso dirigir la agudeza de la mente.

René Descartes

Este capítulo presenta nuestra arquitectura de aplicaciones educativas basadas en videojuegos. Empieza marcando cuáles son sus objetivos, para pasar a mostrar la visión general de la misma. Posteriormente analiza cada uno de los módulos en detalle. El capítulo termina con una evaluación. Más adelante, en el capítulo siguiente se describirá una aplicación educativa desarrollada con ella.

5.1. Introducción

En el capítulo anterior detallábamos la metodología a seguir para el desarrollo de aplicaciones educativas basadas en videojuegos. Veíamos que la división del contenido en dos partes claramente diferenciadas, entornos virtuales y contenidos pedagógicos (ejercicios), es beneficiosa en muchos aspectos.

En este capítulo pasamos a describir la arquitectura software que soporta esa metodología. Como veremos, las partes principales de las que se compone una de estas aplicaciones son las mismas ya disponibles por separado en los ITS y los videojuegos.

En realidad, ya en el capítulo anterior describíamos una arquitectura muy general. En aquel momento, al hablar de los contenidos dividimos la aplicación en dos grandes bloques: el "motor de la aplicación" y el "código

específico". La figura 4.3 reflejaba que el primero de ellos lee los datos relacionados con el entorno virtual, mientras que el segundo se encarga de la información relacionada con el dominio educativo.

Tras una enumeración de los objetivos que perseguimos con la arquitectura, el capítulo pasa a refinar el esquema anterior, separando esos dos grandes bloques en otros sistemas más pequeños, agrupando la funcionalidad y minimizando la dependencia entre ellos.

Una vez descrito el esquema general refinado de la arquitectura, se pasan a detallar cada uno de los módulos. Posteriormente, se explica de qué forma la arquitectura soporta la separación entre los dos tipos de información (entorno virtual y conocimiento del dominio) que propone la metodología del capítulo 4. El último apartado del capítulo realiza una evaluación.

5.2. Objetivos de la arquitectura

Los objetivos principales que guían el diseño de la arquitectura expuesta en este capítulo son los siguientes:

1. Minimizar la dependencia entre el conocimiento específico del dominio que se enseña y el resto del sistema. De esta forma, el conocimiento del dominio puede aprovecharse para implementar otros videojuegos educativos que enseñen lo mismo pero de distinta forma. También permite utilizar las partes no específicas del dominio en varios juegos educativos. En definitiva, permite la *reutilización*, de forma que se reduzcan los costes de creación de este tipo de aplicaciones.

2. Permitir que en desarrollos posteriores sean fácilmente sustituibles algunos módulos. Principalmente se perseguirá poder *intercambiar* fácilmente aquellos módulos susceptibles de quedar obsoletos con el avance tecnológico.

3. Permitir que los distintos profesionales que entran en juego durante el periodo de desarrollo puedan trabajar sin solaparse unos con otros, minimizando las dependencias entre ellos.

Todos estos objetivos pueden resumirse en uno: reducir al mínimo la dependencia entre distintas partes, para hacerlas intercambiables.

Para ello, la arquitectura parte de una premisa: la metodología de creación de contenidos utilizada para el desarrollo *debe* ser la expuesta en el capítulo anterior. Es decir, los ficheros de datos que alimentarán a la aplicación deben estar claramente divididos en los que sirven para almacenar las propiedades del entorno virtual, y los utilizados para guardar el contenido educativo.

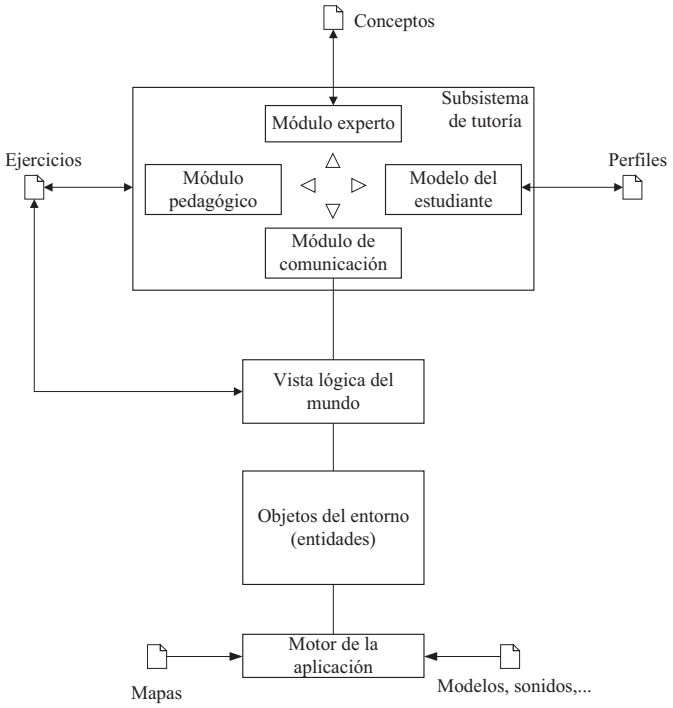


Figura 5.1: Arquitectura general de un sistema educativo basado en videojuegos

5.3. Visión general

La figura 5.1 muestra un esquema general de la arquitectura. A primera vista, podemos distinguir dos grandes bloques: el encargado de la gestión del sistema de tutoría, y el responsable de la gestión del entorno virtual.

El primero de ellos (parte superior de la figura) se basa en las ideas que describimos en el capítulo 3. El subsistema de tutoría está formado por tres grandes módulos encargados de tomar las decisiones pedagógicas, en base al perfil o modelo del estudiante y al dominio concreto que se está enseñando. Estos módulos necesitan una serie de datos que normalmente se encuentran contenidos en ficheros: los conceptos o conocimiento del dominio, los ejercicios y el perfil del estudiante.

El *módulo de comunicación* del subsistema de tutoría es el que sirve como punto de unión entre todo el subsistema y el exterior; todos los módulos internos deben pasar por el de comunicación para interactuar con el entorno.

El resto de subsistemas, que aparecen en la parte inferior de la figura, están relacionados con el entorno virtual. El motor de la aplicación es el de más bajo nivel, encargado de tareas como dibujar o leer el estado de los dis-

positivos de entrada. También es el encargado de leer los mapas que definen el aspecto visual. Durante el capítulo anterior, esta componente aparecía en los diagramas bajo el nombre de *motor del juego*.

Por encima de éste, aparecen los objetos o entidades del entorno. En la sección 2.6 definíamos a las entidades como "una pieza autocontenida de contenido interactivo" (Llopis, 2005b), definición que puede aplicarse tanto en videojuegos como en cualquier aplicación que utilice un entorno virtual. Podemos ver este módulo como una capa que gestiona la apariencia o comportamiento dentro del interfaz de los objetos interactivos y autónomos. Por ejemplo, en esta capa se coloca el funcionamiento de puertas, interruptores, ascensores y otros elementos del entorno.

En videojuegos, es habitual llamar a ese módulo el *código específico del juego* (ver sección 2.4 y figura 4.2). En esas aplicaciones, la inteligencia artificial suele ser lo suficientemente simple como para poder codificarla directamente en este subsistema. En nuestro caso, situamos la *vista lógica* en otro módulo distinto, que almacena la parte del estado del mundo relevante desde el punto de vista educativo. Gracias a eso, el subsistema de tutoría tiene toda la información que necesita, y por lo tanto esta *vista lógica* se convierte en el único punto de unión entre él y el entorno virtual. Como veremos, en este módulo también implementaremos la inteligencia artificial de más alto nivel de algunos de los objetos del entorno.

En los apartados siguientes analizamos en detalle cada uno de estos módulos.

5.4. Subsistema de tutoría

El subsistema de tutoría sigue el esquema visto en la sección 3.2. Los cuatro módulos que aparecen en la figura 5.1 son los siguientes:

- Modelo del estudiante: contiene una representación del estudiante. Esta representación suele incluir una lista de los conceptos que aparentemente conoce, los ejercicios que ha resuelto previamente, o ciertos parámetros que definen su personalidad. Durante la ejecución de la aplicación, el modelo del estudiante es actualizado con la información recibida desde el resto de módulos del subsistema de tutoría. Eso incluye las posibles situaciones que se pueden dar en el entorno virtual y que llegan a través del módulo de comunicación.

- Módulo pedagógico: es el que decide qué ejercicio presentar al usuario a continuación, por lo que tiene que tener acceso a la base de datos de ejercicios. También tiene otras tareas relacionadas con la parte educativa. Por ejemplo, si existe una monitorización continua del estudiante, se encarga de emitir al módulo de comunicación las órdenes necesarias

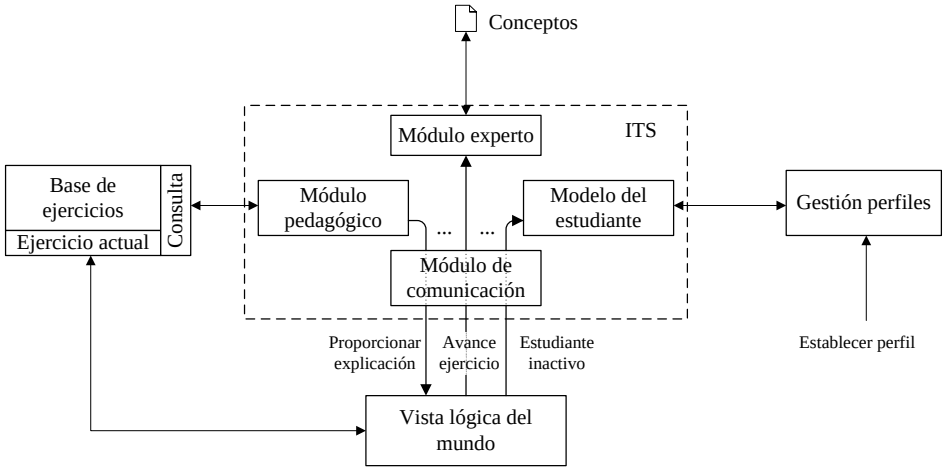


Figura 5.2: Vista detallada del subsistema de tutoría

para que el interfaz haga llegar al usuario las explicaciones y sugerencias que parece necesitar.

- Módulo experto: es el que contiene el conocimiento del dominio, consistente en los conceptos que hay que enseñar, explicaciones, y modo de resolver los ejercicios. Si el módulo no es lo suficientemente sofisticado y no es capaz de resolverlos, las soluciones deberán estar almacenadas en la propia base de problemas, y el módulo experto leerá de ella la solución. En ese caso, la principal tarea del módulo es comparar la solución parcial aportada por el estudiante con la leída, y detectar cuando las discrepancias son excesivas.

- Módulo de comunicación: sirve de interfaz entre el subsistema de tutoría y el resto de la aplicación. La comunicación se realiza en base a la vista lógica del mundo, y es por lo tanto una comunicación de alto nivel. No se tienen pues en cuenta detalles como animaciones de los avatares o colisiones del personaje con el entorno, sino acciones relevantes para el módulo de tutoría, como progreso en la ejecución del ejercicio.

Aunque los módulos aparecen separados en el esquema general, en realidad todos ellos están comunicados, e intercambian información para realizar su labor. Además, todos interactuarán en mayor o menor medida con la parte "visible" de la aplicación, modificando ciertos aspectos de ésta, a través del módulo de comunicación, y recibiendo información sobre los cambios que se producen, para actualizar su visión de ella. En la figura 5.2 se pueden ver algunos ejemplos de la información que fluye a través del módulo de comu-

nicación. En particular, el módulo pedagógico puede decidir que es preciso dar alguna explicación al usuario, y así puede hacérselo saber a la vista lógica del mundo; la vista lógica puede informar del avance en la ejecución del ejercicio, para que el módulo experto compruebe si el estudiante va bien o no; también puede indicar que el estudiante lleva ocioso mucho rato sin hacer nada realmente interesante en el entorno, por si el subsistema de tutoría quiere tomar alguna medida.

Además de compartir información entre ellos, el esquema general de la arquitectura establece que los módulos pueden utilizar datos externos como fuente de información.

En este sentido, el módulo experto puede leer de disco por ejemplo los conceptos que se pretenden enseñar, o cualquier otro tipo de información que le ayude a resolver los ejercicios que se plantearán al usuario, así como explicaciones o textos que se le proporcionan.

El módulo pedagógico, por su parte, debe tener acceso a la base de ejercicios. En nuestra arquitectura, en vez de ser el propio módulo el encargado de leer de disco la lista de ejercicios para decidir cuál es el más adecuado, delegamos esa responsabilidad en un módulo externo (que no aparece en la figura 5.1). Este módulo lee todos los ejercicios y tiene el concepto de "ejercicio actual". El módulo pedagógico puede pedir al módulo, a modo de consulta en una base de datos, que seleccione el ejercicio que ponga en práctica una serie de conceptos, que éste habrá seleccionado previamente de acuerdo al perfil del estudiante. El ejercicio seleccionado, será también utilizado por el módulo responsable de la vista lógica del mundo, según veremos en la sección 5.5. Siguiendo esta aproximación, el módulo de recuperación de ejercicios puede aplicar técnicas sofisticadas, como las de adaptación nombradas en la sección 4.4 que, en vez de devolver ejercicios existentes en la base de datos, creaba ejercicios nuevos ajustados a la consulta, adaptando los ya existentes (Gómez Martín, Gómez Martín, Díaz Agudo y González Calero, 2005c).

Por último, el módulo responsable de mantener actualizado el perfil del usuario, también utiliza un módulo externo encargado de la base de datos de todos los estudiantes. En esos ficheros está la información de todos los usuarios del sistema. Cuando se inicia la aplicación, el interfaz gráfico se encarga de que el usuario se identifique en el sistema, para poder cargar su perfil. Eso provoca que el módulo gestor de los perfiles active el estudiante actual (ya sea creando un nuevo perfil, o cargando de disco el existente), para que el modelo del estudiante del subsistema de tutoría lo tenga disponible. Este tipo de comportamiento es el funcionamiento habitual en las aplicaciones de entretenimiento, donde un jugador, al arrancar la aplicación, "carga" una partida que dejó a medias en la ejecución anterior.

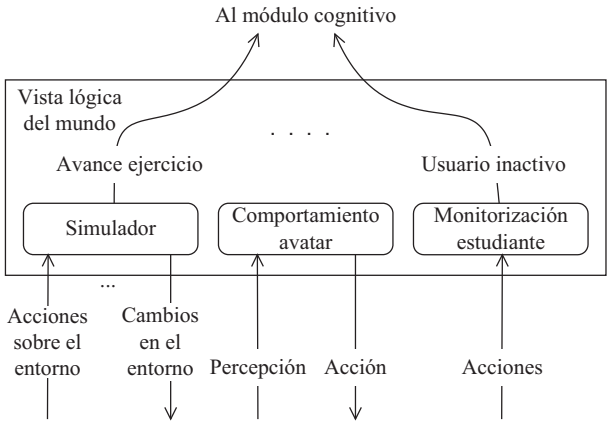


Figura 5.3: Esquema detallado del módulo de vista lógica

5.5. Vista lógica del mundo

Para que sea posible el razonamiento que se realiza en el subsistema de tutoría, es necesario que la representación del estado del entorno virtual no sea demasiado específica. Lo mismo ocurre para poder definir el comportamiento inteligente de los objetos del entorno más importantes desde el punto de vista del sistema educativo.

En este módulo se almacena el estado del mundo, a un nivel de abstracción suficiente como para poder aplicar las técnicas de razonamiento adecuadas. Puede estar codificado en el lenguaje de programación utilizado en el resto de la aplicación, o puede hacer uso de otro tipo de técnicas, como sistemas expertos u ontologías.

En cualquier caso, la visión lógica del mundo sirve de *fachada* que oculta los aspectos de más bajo nivel del entorno virtual que dificultarían ese razonamiento, de tal forma que todo lo que el subsistema de tutoría necesita saber para realizar su función se encuentra disponible en este módulo.

Para funcionar, recibe información de bajo nivel del entorno virtual y la abstrae para almacenar únicamente la parte relevante al subsistema de tutoría. Por tanto, sirve de *filtro* que aisla de la batería de hechos que ocurren en el entorno virtual los que son realmente interesantes desde el punto de vista pedagógico.

En este módulo se desarrollan tres funciones principales (ver figura 5.3):

- Simulación: para todas aquellas aplicaciones educativas que simulan algún aspecto de la vida real (como el manejo de una sala de máquinas) el código responsable de esa simulación está ubicado aquí. De esta forma, el módulo almacena el estado del mundo. Cuando se ejecutan

ciertas acciones en el entorno virtual (como activación de interruptores por parte del estudiante), éstas provocan cambios en la visión lógica, que es actualizada por el simulador. Estas modificaciones, a su vez, son propagadas de nuevo al entorno virtual, para que se actualice su apariencia. El avance del simulador, además, se redirige al módulo experto del subsistema de tutoría, ya que es, en última instancia, el que conoce cómo se debe resolver el ejercicio/situación al que se está enfrentando el estudiante. A la vista de los datos, y con la ayuda del módulo pedagógico, el sistema experto puede decidir parar al usuario para informarle sobre errores, darle pistas, o permitirle continuar sin realizar ninguna acción especial.

- Inteligencia artificial de alto nivel: algunos de los objetos que aparecen en el entorno virtual son relevantes al contexto educativo utilizado. Si, por ejemplo, estamos en un entorno de simulación de un incendio para entrenamiento de bomberos, las actuaciones del resto de bomberos y víctimas (que en el entorno aparecen como avatares) son muy importantes, ya que influyen en el tipo de conceptos que el estudiante aprenderá; dependiendo de su comportamiento, el entorno puede ayudar a aprender a cómo reaccionar ante situaciones de pánico del resto de compañeros, o a desalojar una víctima que está paralizada por el miedo. La inteligencia artificial de estos *objetos relevantes* desde el punto de vista educativo se encuentra en este módulo, ya que dependen fundamentalmente del ejercicio actual y de lo que dicte el subsistema de tutoría.

- Monitorización del estudiante: la parte más importante de la monitorización del estudiante es conocer qué acciones está realizando sobre el entorno virtual, y eso ya se hace gracias a las dos funcionalidades anteriores. No obstante, debemos hacer mención explícita de esta tarea, pues existe otro tipo de monitorización que no tiene por qué estar relacionada con las acciones sobre el entorno virtual. Por ejemplo, puede ser relevante saber si el estudiante lleva mucho tiempo inactivo o si su avatar está mirando al personaje que está dándole una explicación. En todos esos casos, este módulo recibe las acciones que el usuario realiza sobre el entorno virtual, e informa al subsistema de tutoría para que, al menos, actualice el modelo del estudiante.

En la figura 5.3 se muestra un esquema de este módulo. Como ya hemos dicho, el simulador recibe algunas de las acciones que se realizan en el entorno (aquellas que tienen que ver con las entidades que representan partes de la estructura simulada), lo que hace que el estado interno del simulador cambie; ese cambio puede generar alteraciones en el aspecto visual de algunas entidades, por lo que el simulador envía esa información a la capa inmediatamente inferior. Por su parte, la inteligencia de alto nivel de los avatares

reciben eventos de percepción, y en base a ellos deciden qué acciones realizar. Por último la monitorización del estudiante recibe las acciones que éste hace, y obra en consecuencia. En la figura, aparecen algunas de las invocaciones que este módulo puede hacer sobre el subsistema de tutoría (aquellas que ya mostrábamos como entradas a él en la figura 5.2), pero hay muchas más, tanto en sentido ascendente como descendente. Por ejemplo, el módulo experto puede pedir al simulador que anule la última acción ejecutada (debido a un error del estudiante) o dar órdenes concretas a un avatar (como veremos en la sección 5.10). En la sección siguiente, además, veremos cómo estructurar el módulo encargado de las entidades del juego para poder crear fácilmente la comunicación con este módulo.

La vista lógica, recibe, al principio de la ejecución, el ejercicio o episodio educativo al que se va a enfrentar el usuario, y se configura dependiendo de sus características. Dada su importancia, dedicamos toda la sección 5.11 a la descripción de cómo tanto este módulo como el entorno virtual son configurados en base al ejercicio cargado.

5.6. Entidades del juego

Ya hemos hablado en la sección 2.6 sobre las entidades de un juego, que forman lo que en la figura 4.2 llamábamos *código específico*. En aquel momento, este *código específico* era "todo lo que no era motor de juego", y por lo tanto constituía la parte más dependiente del juego particular.

En una aplicación educativa, las entidades del juego siguen representando los objetos *dinámicos* del entorno virtual (ya sean visibles o invisibles), que reaccionan ante distintos eventos provocados por el propio estudiante u otras entidades. Son, en definitiva, las que proporcionan la sensación de credibilidad, de entorno *real*, al usuario. La diferencia estriba en que ahora este módulo no constituye el *final* de la implementación, ya que por encima reside, como hemos visto, el subsistema de tutoría apoyado por la vista lógica del entorno.

Igual que en los videojuegos, las entidades son creadas desde el motor de la aplicación que es quien lee el mapa con la información de todo el entorno virtual. Para poder crear entidades (objetos) que están en otro módulo distinto y mantener, aún así, la independencia entre ambos, se utiliza alguna de las técnicas explicadas en la sección 2.6.2, como factorías extensibles o uso de tablas de símbolos de librerías dinámicas. El motor de la aplicación fija la forma en la que lo hará, y el módulo de las entidades seguirá ese convenio.

Podemos distinguir dos tipos de entidades, desde el punto de vista de la aplicación educativa:

- Aquellas entidades que no son específicas del contexto educativo: En este grupo se incluyen todas las entidades genéricas que pueden ser

reutilizadas entre distintos proyectos. Pertenecen a este grupo los elementos arquitectónicos dinámicos habituales: puertas, plataformas o ascensores. También aparecen aquí interruptores, fuentes de sonido, o personajes secundarios no influyentes en el desarrollo del aprendizaje.

- Entidades específicas del contexto educativo: son todas aquellas entidades que influyen en mayor o menor medida en el proceso de aprendizaje. Podemos dividir este tipo de entidades en los mismos tres grupos en los que dividíamos la funcionalidad del módulo anterior (sección 5.5). Así, tenemos:

  - Entidades que representan los objetos gestionados por el simulador. Un ejemplo, como veremos en el capítulo siguiente, son las que en JV^2M representan las estructuras de la máquina virtual de Java (JVM). Para mantener la coherencia entre el estado en el simulador y el mostrado en el entorno, se puede hacer uso del patrón observer (Gamma et al., 1995), o buscar alguna otra alternativa de actualización.
  - Entidades que son lo suficientemente relevantes en el contexto pedagógico como para que deban ser accedidas por el subsistema de tutoría. En ese caso el responsable de su funcionamiento, o inteligencia de alto nivel, no está en este módulo sino en la vista lógica del apartado anterior. Estas entidades se encargan únicamente del aspecto visual en el entorno virtual, y de su inteligencia de bajo nivel (búsqueda de caminos o percepción básica). La vista lógica recibirá eventos de percepción, y podrá preguntar por ciertos aspectos del entorno, y ejecutar primitivas en él para cambiar el aspecto visual. Veremos en la sección 5.7 una lista posible de las acciones que estas entidades deben hacer públicas al módulo superior para poder ser controladas desde fuera.
  - Entidad que monitoriza al estudiante. Aunque en los juegos no es habitual que exista una entidad que represente al jugador (especialmente en los de *primera persona*), en nuestra arquitectura situamos una entidad específica para el estudiante. Ésta es la encargada de informar a su parte contraria en la vista lógica, de tal forma que se pueda, en última instancia, actualizar el modelo del estudiante.

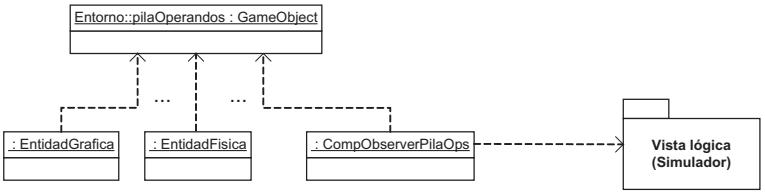
Para la implementación se puede utilizar el mecanismo de herencia habitual, de tal forma que exista una clase base para todos los objetos del juego, del que van derivando el resto de clases. La herencia permite agrupar funcionalidad común de varias clases, en la clase padre única. Un ejemplo de jerarquía de entidades puede verse en la figura 2.4.

Nosotros, no obstante, aconsejamos una implementación basada en componentes (sección 2.6.3). Esta aproximación convierte a cada objeto del entorno virtual en una lista de componentes con un interfaz común, pero con funcionalidad distinta. Cada componente está encargado de una parte específica de la entidad, por ejemplo, de su representación gráfica, de la gestión de sus colisiones, o de su nivel de vida. La comunicación entre los componentes de la entidad se hace a través de mensajes genéricos. Un ejemplo con este escenario podría ser el siguiente: el motor de colisiones envía un evento de tal forma que el componente encargado de las colisiones es informado de una colisión entre el objeto del juego y, digamos, una bala; éste envía entonces un mensaje a todos los componentes del objeto para que se den por enterados; el componente encargado del nivel de vida disminuirá ésta en la proporción que considere oportuno.

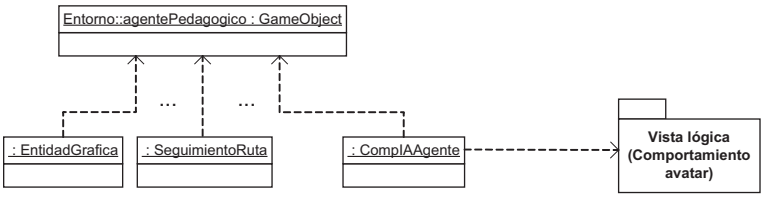
Este modo de actuar facilita mucho la programación, ya que los objetos pasan a ser un conjunto de componentes configurables, de tal forma que la creación de objetos nuevos consiste en decidir cuáles de esos componentes ya disponibles deben pasar a formar parte del objeto y con qué parámetros. Sólo en algunos casos se necesitará la programación de componentes específicos para algunos objetos del juego.

Si la arquitectura de entidades basadas en componentes supone un avance para la programación de videojuegos (ver sección 2.6.3), en el caso de aplicaciones que siguen nuestra arquitectura suponen una ventaja adicional. Como hemos dicho anteriormente, la diferencia entre las entidades de un videojuego y de una aplicación educativa basada en ellos es la existencia de objetos que tienen que ver con el simulador, con el proceso de aprendizaje y con el estudiante. Además, algunas de las entidades pueden pertenecer a varios de los grupos anteriores. Por todo esto, el mecanismo de herencia no es suficiente, ya que únicamente permite un eje de variabilidad en cada nivel de la jerarquía de clases. La solución es el uso de componentes que dividen la funcionalidad en distintas clases de tamaño reducido, de tal forma que una entidad está compuesta por una lista variable de componentes. Utilizando este esquema, la programación de las entidades sería:

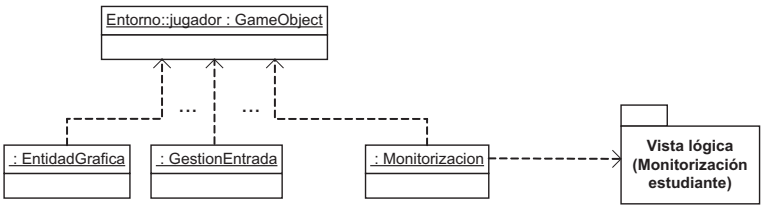
- Para las entidades que tienen su contrapartida en el simulador, se puede añadir un componente especializado, encargado de comunicarse con él. En la figura 5.4a aparece un esquema de una situación en tiempo de ejecución de JV^{2}M, el sistema descrito en el capítulo 6. El objeto del entorno `pilaOperandos` que representa una pila de operandos de la máquina virtual de Java, está compuesta por tres componentes (en la práctica contiene más; en la figura únicamente se muestran esos tres). El primero simboliza la representación gráfica de la pila de operandos, el segundo la encargada de la física/colisiones y el tercero, el específico, encargado de la comunicación con el simulador (en el módulo de la



(a) Ejemplo de entidad que representa a otra en el simulador



(b) Ejemplo de entidad que representa el avatar de un agente pedagógico



(c) Ejemplo de entidad para representar al estudiante

Figura 5.4: Ejemplos de entidades basadas en componentes

vista lógica). Esta comunicación es bidireccional. Cuando un avatar interactúa sobre el objeto del entorno, el simulador es informado; cuando el simulador cambia de estado, informa al objeto para que cambie su apariencia.

- Para las entidades cuyo comportamiento está definido en la vista lógica, puede añadirse, igual que antes, un componente dedicado que se encarga de la comunicación. Dado que la vista lógica únicamente está interesada en información de alto nivel, el propio componente puede encargarse de filtrar todo lo que ocurre en el entorno virtual, y enviar únicamente la información relevante. La vista lógica, que implementa el funcionamiento, puede utilizar el mismo componente como elemento de comunicación, invocando la ejecución de ciertas primitivas de alto nivel que son traducidas después por el propio componente a primitivas sobre el entorno virtual por ese componente. En la figura 5.4b aparece un ejemplo parcial del objeto que podría representar al avatar de un agente pedagógico en el sistema (ver sección 5.10 para más detalles). Dispone de un componente encargado de su representación gráfica y un componente capaz de hacer mover al avatar por una ruta calcula-

da. El último de los componentes que aparece es el que se encarga de recibir las órdenes de comportamiento de la vista lógica y traducirlas a mensajes de acciones primitivas, y de enviar a ésta los eventos de alto nivel que considere oportunos.

- De manera similar a la de las entidades anteriores, la entidad encargada de gestionar las acciones del estudiante tiene componentes para la gestión de la entrada, que moverán su avatar de acuerdo a ella (ver figura 5.4c). Si el entorno no es en primera persona, también habrá al menos un componente más encargado de la parte gráfica. Con este escenario, se puede añadir un componente adicional, que se encargue de enviar las señales oportunas a la vista lógica del mundo cuando ocurra un evento digno de ser conocido por el subsistema de tutoría.

Por último, si existe alguna entidad en el entorno virtual que juegue varios de los papeles anteriores, la entidad estará formada por los dos componentes correspondientes, por lo que el esfuerzo de programación se minimiza.

5.7. Comunicación entre vista lógica y entidades

En la sección anterior veíamos que la comunicación entre el módulo que representa la vista lógica del sistema y el de entidades puede dividirse en tres partes: la comunicación entre las entidades y el simulador, entre ciertas entidades y su inteligencia artificial de alto nivel, y entre el avatar del estudiante y el módulo encargado de su monitorización. Las tres conexiones se hacían explícitas en la figura 5.4.

La comunicación que se intercambia entre el avatar del estudiante y la vista lógica es escasa, y no requiere de mucho esfuerzo de diseño. El componente encargado de él en el entorno virtual simplemente invoca las primitivas disponibles en la vista lógica.

En cuanto a la comunicación entre las entidades del entorno y los objetos del simulador, se puede hacer uso del patrón observer, de tal forma que cuando el simulador cambia su estado interno, el objeto correspondiente encargado de su representación es informado. En particular el componente que en la figura 5.4a llamábamos `CompObserverPilaOps` puede jugar perfectamente ese papel. Cuando el componente es invocado desde el simulador, éste envía los mensajes internos necesarios al resto de componentes para provocar el cambio de apariencia.

En dirección contraria, cuando algún avatar interactúa con el entorno, la entidad correspondiente, deberá invocar las primitivas de cambio de estado del simulador, para que éste refleje la nueva situación, e informe al entorno virtual qué cambios de apariencia se deben hacer.

Por su parte, hemos dicho que las entidades relevantes desde el punto de vista pedagógico, tienen su comportamiento codificado en dos sitios distintos. Por una parte, las acciones de bajo nivel, como la búsqueda de rutas, la reproducción de frases, o la interacción con otras entidades se sitúa en el módulo de objetos del entorno. Por otra parte, la inteligencia de alto nivel, que decide dónde tiene que moverse el avatar, qué frases debe decir o con qué entidades debe interactuar, se sitúa en la vista lógica del mundo.

En principio, podríamos forzar a que el módulo de alto nivel base sus decisiones únicamente en el estado del resto de entidades que él mismo maneja, o en el estado del simulador. En contextos simples este escenario puede ser suficiente. Sin embargo, en un esquema general como el nuestro, delimitar de esta forma la información con la que la vista lógica cuenta sería restringir demasiado sus posibilidades de actuación.

Por esta razón, definimos una serie de *perceptores* que son enviados por el componente correspondiente de la entidad del juego desde el entorno virtual hacia el responsable de la toma de decisiones de la vista lógica. También se establece una lista de *actuadores* con los que la vista lógica podrá alterar el entorno virtual. Por último, la inteligencia artificial podrá hacer consultas generales sobre el entorno, que también deberán ser contestadas por el componente en cuestión.

El conjunto de primitivas puede variar significativamente dependiendo de la temática de juego elegida. Para minimizar esa dependencia, la lista de perceptores y actuadores debe ser lo suficientemente general como para que sea útil al menos para una familia de temáticas, por ejemplo aquellas que se basan en las mecánicas de una aventura gráfica, o en un juego en primera persona.

No obstante, aún en el caso de necesitarse el cambio de temática de una aplicación educativa concreta, la arquitectura facilita la migración. Esto es debido a que la implementación de la comunicación entre el entorno virtual y la vista lógica está *aislada* en un componente específico de la entidad. Como componente que es, recibirá todos los mensajes internos que llegan a él, y en particular, todos los relacionados con la *percepción*. Además, también puede enviar mensajes internos a sus componentes hermanos. Por lo tanto si las acciones y percepciones que la vista lógica espera cambian, únicamente habrá que sustituir el componente encargado de la comunicación por otro actualizado.

En la tabla 5.1 aparece una lista de acciones razonable para una aplicación basada en las mecánicas de una aventura gráfica. Muchas de estas acciones tienen como objeto de la acción otra entidad distinta. Por ejemplo, la acción `seguirA` tiene como "parámetro" a qué entidad se debe perseguir. En vez de utilizar esos parámetros, definimos el concepto de *entidad objetivo* de otra entidad, de tal forma que la acción `seguirA` utiliza como destino esa entidad objetivo. Para cambiar la entidad objetivo, existe una acción

Perceptores	
`estudianteVisto`	Evento que indica que el estudiante está en el ángulo de visión de la entidad.
`entidadUsada`	Evento que indica que la entidad bajo control ha sido *utilizada* por otra entidad
`entidadCogida`	Evento que indica que la entidad bajo control ha sido *cogida* por otra entidad
`entidadUsadaCon`	Evento que indica que la entidad bajo control ha sido *utilizada* en combinación con otra entidad.
Actuadores	
`seleccionaObjetivo`	Establece el objetivo de la entidad, de acuerdo a un nombre.
`irA / seguirA`	Órdenes de movimiento hacia la entidad objetivo.
`mirarA`	Encara el avatar hacia la entidad destino.
`hablarA`	Dice una frase mirando hacia la entidad objetivo.
`usar / usarCon`	Operaciones para manipular una entidad, o una entidad con otra.
`coger`	Coge un objeto del entorno, y lo guarda en el inventario.
`seleccionarInventario`	Selecciona un elemento del inventario; al avatar le aparecerá en las manos.

Tabla 5.1: Acciones primitivas entre el entorno virtual y la vista lógica

específica.

La base de todas las acciones es la interacción típica de las aventuras gráficas, donde los avatares (jugador u otros NPCs) pueden interactuar con el resto de entidades utilizando principalmente cuatro operaciones, cuya semántica concreta dependerá de la entidad particular:

Mirar: cuando el estudiante mira a una entidad, aparece una descripción de ésta.

Coger: un avatar puede acercarse a un objeto en el entorno virtual y recogerlo, pasando a formar parte de su *inventario*.

Usar: utilizar una entidad que hay en el entorno, por ejemplo un interruptor. La semántica concreta de la operación dependerá de la propia entidad. Por ejemplo, utilizar un interruptor no tiene el mismo resultado que utilizar una silla.

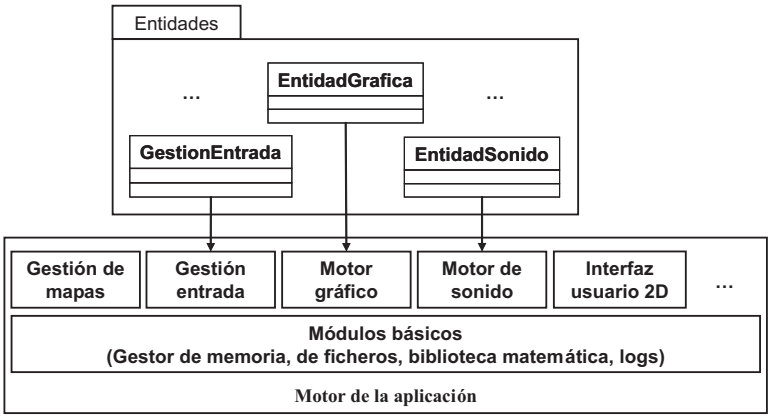


Figura 5.5: Vista detallada del motor de la aplicación

Usar con: se trata de una operación combinada, donde se utiliza una entidad del inventario con una entidad del entorno. Por ejemplo, se puede utilizar una bombilla en el inventario con un casquillo en el entorno.

Por otro lado, además de la lista de perceptores y actuadores, es posible que la parte encargada de la inteligencia artificial necesite conocer ciertos aspectos del estado de bajo nivel del entorno, como por ejemplo, la lista de entidades que entran dentro del campo de visión o la lista de entidades que pertenecen a un cierto tipo. Esas funciones también las puede implementar el componente, que delega su implementación en algún otro elemento interno del módulo.

5.8. Motor de la aplicación

El motor de la aplicación es el módulo de más bajo nivel de toda la arquitectura. Se encarga de las tareas más específicas de la plataforma en la que se ejecuta el programa, y sobre las que se apoyan el resto de los módulos que, gracias a éste, podrán programarse de forma independiente del sistema operativo y máquina subyacente.

Aunque en el esquema general de la figura 5.1 el módulo aparece como un bloque indivisible, en realidad está compuesto por distintos subsistemas, todos ellos sustentados sobre una serie de utilidades básicas (ver figura 5.5):

- Gestor de memoria: utilizado para pedir y liberar bloques de memoria dinámica. Disponer de un gestor propio facilita la detección de fugas de memoria por la no liberación de punteros, y la monitorización de la cantidad de bloques que se piden por unidad de tiempo.

- Gestor de ficheros: sirve de fachada al sistema de archivos real. La ventaja es que su comportamiento puede cambiarse de forma transparente al resto de la aplicación. En la fase de desarrollo, el gestor *monitoriza* el uso que se hace de los ficheros, comprobando el orden de acceso y otros parámetros. Esa información se utiliza posteriormente para mejorar el rendimiento de las operaciones de lectura, por ejemplo grabando seguidos en el soporte físico los ficheros que se leen de forma consecutiva. También permite empaquetar todos los archivos en un único fichero comprimido, de tal forma que el propio gestor lo va descomprimiendo de forma transparente al resto de la aplicación.

- Librería matemática: contiene la implementación de las funciones que normalmente son utilizadas en los subsistemas superiores del motor de la aplicación. Por ejemplo, dispone de funciones para el tratamiento de vectores, matrices, interpolaciones, etc. La existencia de este módulo en la parte más baja de la arquitectura se debe a que su implementación puede aprovechar las características del *hardware* concreto sobre el que se ejecuta la aplicación (instrucciones MMX, SSE o 3DNow! entre otros).

- Sistema de logs: se utiliza con fines de depuración. En vez de escribir en pantalla directamente, el programador utiliza el sistema de logs para escribir información de depuración. El sistema permite indicar parámetros del mensaje, como el origen, el grado de importancia, etc. El sistema de logs puede estar implementado de tal forma que escriba todos estos mensajes en pantalla, o puede ser más sofisticado y presentarlos en una ventana auxiliar que permite filtrarlos por categorías o incluso enviarlo por red a una máquina remota.

En realidad, todas las utilidades básicas descritas sirven como punto de entrada o fachada a alguna utilidad básica del propio lenguaje o de su biblioteca estándar (memoria dinámica, operaciones aritméticas, entrada/salida a disco y escritura en el terminal). Gracias a la existencia de esta fachada, se puede mejorar la funcionalidad, sustituyendo la implementación por una hecha a medida, con fines de depuración u optimización. Por lo tanto, para sacar el máximo partido a estos módulos, el resto de sistemas de la aplicación deben hacer uso de ellos, en vez de seguir utilizando las funcionalidades estándar. En la práctica, sin embargo, al perseguir la máxima independencia entre los módulos, estos sistemas base suelen utilizarse, a lo sumo, en el módulo inmediatamente superior, el responsable de las entidades del entorno virtual. Romper la independencia y *conectar* los niveles más abstractos de la arquitectura (vista lógica del mundo y subsistema de tutoría) para utilizar estos módulos básicos sería difícil de justificar.

El resto de subsistemas del motor de la aplicación se construyen sobre estos cuatro pilares básicos. Cada uno de ellos proporciona una funcionalidad

diferente, como la reproducción de un sonido, o el dibujado de un modelo tridimensional. Casi para cada uno de estos módulos existe, en el módulo de entidades visto en la sección 5.6 un componente específico que se comunica con él. Si un objeto del entorno virtual tiene una representación gráfica, poseerá (como veíamos en la figura 5.4), un componente `EntidadGrafica`, que será el que se comunique con el motor gráfico. De manera similar, existe un componente para la gestión del sonido, colisiones, física o entrada de usuario. En la figura 5.5 queda reflejado este hecho; como se ve, en la parte superior aparecen distintas clases (componentes) del módulo de gestión de entidades, cada una haciendo referencia a un subsistema específico.

Procediendo de esta forma, volvemos a minimizar el impacto que tendría la modificación de uno de estos subsistemas. Si se cambia un módulo por otro más sofisticado, únicamente será necesario sustituir el componente concreto que lo manejaba en el módulo inmediatamente superior. Gracias a la arquitectura basada en componentes, ese cambio es aprovechado automáticamente por todas las entidades que utilicen ese componente.

Los módulos que aparecen en el motor de la aplicación dependen de la complejidad de esta. Los habituales son:

- Motor gráfico: es el responsable del dibujado tridimensional. También se encarga de la creación de la ventana, utilizando las llamadas al sistema operativo subyacente.

- Gestión de entrada: tanto de teclado y ratón como de otros dispositivos menos habituales, como palancas de juego (*joysticks* y *gamepads*) y volantes. Su funcionamiento puede depender del motor gráfico, ya que en ocasiones los eventos de entrada son recibidos por el creador de la ventana sobre la que se realiza la entrada.

- Interfaz de usuario: se encarga de la presentación de elementos en dos dimensiones, como botones, etiquetas o listas de opciones. Es utilizado por la aplicación en los menús, información durante el juego, HUDs, etc.

- Motor de sonido: encargado de la reproducción de los distintos tipos de sonido (música, efectos, voces). En sistemas avanzados, éste puede incluir capacidad de síntesis de voz.

5.8.1. Dependencia de datos

El motor de la aplicación es el encargado de leer gran parte de los datos necesarios para la ejecución de la aplicación. En particular, casi cualquiera de los subsistemas requieren el uso de formatos de fichero específicos para su funcionamiento.

Así, por ejemplo, el motor gráfico debe leer ficheros con los modelos tridimensionales, animaciones, texturas y efectos visuales; el motor de sonido los ficheros de ondas; el gestor de la entrada la configuración de los dispositivos, y el del interfaz del usuario las posiciones en pantalla de los elementos.

Esto hace que cada uno de los subsistemas establezcan un *formato de los ficheros de entrada*. En los casos en los que es posible, se utilizan formatos estándar, como por ejemplo para las texturas o ficheros de audio. Sin embargo, para otra gran cantidad de datos, esos formatos son específicos del subsistema escogido. Por ejemplo, los modelos tridimensionales de los personajes están almacenado en formatos distintos si trabajamos con el motor gráfico de Nebula 2 o si lo hacemos con Ogre. Eso mismo ocurre con la carga de mapas que trataremos en la sección 5.8.2. El formato concreto que se utiliza es fijado por el motor de la aplicación.

Para minimizar las consecuencias de la sustitución de alguno de estos módulos por otro distinto y el problema asociado del cambio de los formatos de los ficheros de entrada, la generación de contenidos debe hacerse lo menos específica posible. Las dependencias fundamentales en cuanto a formatos creadas por estos módulos son tanto los modelos tridimensionales como los mapas. Para ambos, deben utilizarse herramientas de autoría independientes del motor de la aplicación, y aplicar herramientas de conversión desde esos formatos generales a los ficheros específicos interpretados por este, como ya indicamos en la sección 4.4. En el apartado 6.7.2 ampliamos esa descripción, utilizando el caso concreto de JV^{2}M, la instanciación más importante de la arquitectura.

5.8.2. Gestión de mapas

La funcionalidad más importante del motor de la aplicación, desde el punto de vista de generación de contenidos, es la *carga de los mapas*, que aparecía ya en la figura 4.2.

Un mapa contiene dos tipos de información, (i) la lista de entidades con sus propiedades (pares atributo-valor), y (ii) los modelos estáticos del entorno y de algunas de las entidades especiales.

A pesar de que la parte más importante es la lista de entidades, existen dos razones fundamentales por las que situamos su carga en este módulo en vez de en el responsable de ellas:

- Los modelos que contienen (información sobre el terreno, paredes de los edificios, escaleras, etc.) son dependientes del motor gráfico, que aparece dentro del motor de la aplicación.

- Las arquitecturas de videojuegos sitúan en sus módulos equivalentes al motor de la aplicación la funcionalidad de la carga de los mapas. Al mantener esa simetría, nuestra arquitectura permite, como veremos en

la sección 5.8.3, utilizar un motor de juego como motor de la aplicación. De esa forma podemos implementar una aplicación educativa basada en videojuegos utilizando, por ejemplo, el motor de un juego comercial como Half-Life o Far Cry.

Por otro lado, debido a que nuestra metodología de creación de contenidos permite configurar los mapas en base a la información recogida en el ejercicio actual, debemos distinguir entre los datos leídos de disco y los datos una vez alterados en base a ese ejercicio. Aunque el método de configuración de unos con otros lo retrasaremos hasta la sección 5.11, es importante hacer notar aquí esa distinción. El módulo responsable de la gestión de mapas debe distinguir entre aquellos que ha *cargado* de disco, y aquellos que ha *instanciado*, es decir, aquellos para los que ha lanzado sus entidades en el módulo de gestión de objetos del entorno y han sido configurados en base al ejercicio actual.

Esto es especialmente importante en aquellas aplicaciones educativas en las que un mismo mapa de disco puede estar instanciado varias veces pero utilizando configuraciones distintas. Un ejemplo de esto sería la aplicación para entrenar a bomberos: cada planta de un edificio en llamas puede estar generado desde el mismo fichero en disco, pero la configuración de cada una de ellas varía según el ejercicio.

Independientemente de si la *instanciación* se hace de un mapa nuevo o de uno que ya se ha usado anteriormente, el gestor debe recorrer toda la lista de entidades que éste posee. Durante el proceso, irá informando al sistema de gestión de objetos para que cree los elementos necesarios para su manejo. Éste es, por lo tanto, otro punto de comunicación entre el motor de la aplicación y el sistema situado inmediatamente por encima de él, por lo que es, potencialmente, un punto de conflicto ante cambios de cualquiera de los dos módulos. La solución de nuestra arquitectura es la siguiente:

- Cada vez que el cargador quiere procesar una entidad, llama a una función genérica de la capa de objetos del entorno, indicando el nombre de la entidad, y sus atributos.

- La capa del entorno, en base al nombre de la entidad actúa en consecuencia.

- En caso de que la entidad necesite acceder a un modelo contenido en el propio mapa, llamará a una función concreta del cargador de mapas, que se encargará de cargar ese modelo en el motor gráfico y devolver una referencia a él.

5.8.3. Implementación

El motor de la aplicación es, sin lugar a dudas, el módulo que más se presta a la incorporación de COTS (ver sección 2.3.1), es decir de módulos comprados a terceros. Es por esto por lo que hemos dedicado una especial atención a mantener la independencia entre sus módulos y el resto del sistema.

Por poner algún ejemplo, para el motor gráfico, se puede utilizar Ogre, Nebula o RenderWare; para el motor de sonido tenemos disponibles fmod, OpenAL o la funcionalidad ofrecida por DirectX, y para la gestión de la entrada, se puede utilizar el API del sistema operativo suybacente, DirectInput o SDL.

Nuestra arquitectura, que basa la programación de entidades en componentes, facilita mucho el intercambio de estos sistemas, ya que minimiza el punto de contacto entre éstos y la gestión de entidades que tiene lugar en el módulo superior.

Una última posibilidad es sustituir por completo el motor de la aplicación por un motor de juego ya existente. En muchos de los juegos comerciales de mayor éxito (Half-Life, Far Cry, Doom, Unreal Tournament, etc.), los estudios que los desarrollan hacen públicos los editores de niveles con los que se crearon los mapas para el juego original, así como el código fuente de la parte de gestión de entidades. De esta forma, los aficionados pueden introducir algunas modificaciones en él. Si la parte liberada es lo suficientemente amplia, la capacidad de modificación del juego original es tal que permite cambiarlo por completo, manteniendo únicamente la parte de más bajo nivel.

Gracias a la división en módulos que hemos utilizado en nuestra arquitectura, es posible implementar una aplicación educativa basada en videojuegos utilizando esa aproximación.

5.9. Control de la ejecución

Decíamos en la sección 2.5 que el bucle principal de la aplicación es el responsable de ordenar la ejecución de cada una de las tareas necesarias para el avance de la aplicación. Para completar la descripción de nuestra arquitectura, resulta pues indispensable indicar dónde situamos esa gestión.

Sin menoscabar el uso de hebras distintas para algunos de los módulos, el bucle principal de la aplicación está situado en el motor de la aplicación que describíamos en la sección 5.8. La razón de esta posición es que el bucle principal es un componente de muy bajo nivel que, además, es extremadamente dependiente de todos los demás subsistemas. En efecto, estos subsistemas suelen tener unas restricciones fuertes en cuando al orden de invocación de sus métodos de actualización, por lo que lo más adecuado es que el componente que define ese orden se encuentre en el mismo sistema.

El bucle principal, además de actualizar todo lo necesario del motor de la aplicación (la lectura de la entrada, dibujado, reproducción de sonidos, etc.), invoca la actualización de los objetos del entorno, situados en el módulo inmediatamente superior (ver figura 5.1).

Para la ejecución de las tareas relacionadas con la vista lógica del mundo, procedemos de la misma forma. Los eventos generados por los objetos del entorno provocan llamadas a la vista lógica que recibe el control para hacer avanzar el estado del mundo o decidir la forma de comportamiento de las entidades bajo su control ante esos eventos.

A su vez, el subsistema de tutoría puede recibir el control por ser invocado desde el simulador (al notar, por ejemplo, que el estudiante lleva inactivo un tiempo superior a un umbral), aunque también recomendamos la existencia de una hebra continua que ejecute tareas de mantenimiento del subsistema.

5.10. Agente pedagógico

Decíamos en la sección 3.5 que un agente pedagógico supone una mejora en el modo de comunicación entre el sistema y el estudiante que lo utiliza. Lo habitual, indicábamos, es que estos agentes vengan representados por un avatar dentro del entorno, que acompaña al estudiante, o aparece cuando se le necesita.

Nuestra arquitectura también permite la incorporación de estos agentes en las aplicaciones, a pesar de no aparecer expresamente en el esquema general de la sección 5.3.

En la implementación de estos agentes suelen existir tres módulos:

- Módulo de percepción: detecta las acciones relevantes que suceden en el entorno virtual.
- Módulo cognitivo: es el que, utilizando algún mecanismo de razonamiento, decide qué acciones realizar.
- Módulo de movimiento: una vez decididas las acciones a ejecutar, es el que se encarga de hacerlas efectivas.

El eje central es el módulo cognitivo, que decide por ejemplo si es conveniente parar al estudiante para proporcionarle explicaciones, dejarle avanzar, o incluso resolver el ejercicio para que éste aprenda la solución. Debido a esto, el módulo está implementado en el subsistema de tutoría descrito en la sección 5.4.

Las decisiones que toma este módulo se basan en tres tipos de información diferentes:

- Avance del ejercicio: es decir, cómo va resolviendo el usuario el problema que tiene entre manos.

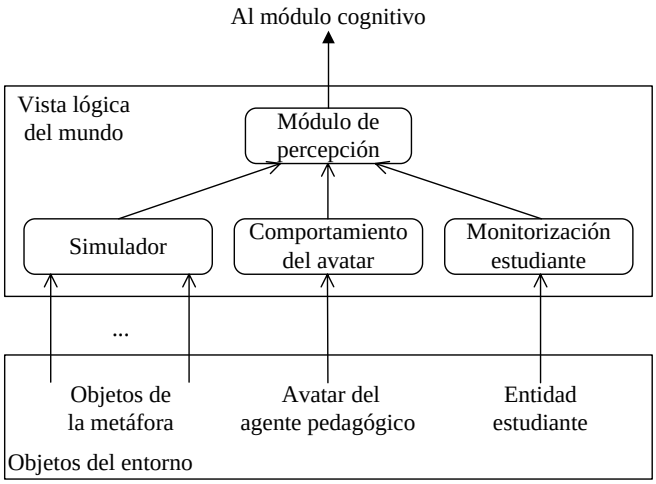


Figura 5.6: Esquema de implementación del módulo de percepción de un agente pedagógico

- Estado del avatar que representa al agente pedagógico: por ejemplo, si tiene bajo su campo de visión al usuario, si éste le está mirando, o le pide ayuda.
- Acciones del estudiante: no sólo todas las que hace (ya sea para hacer avanzar el ejercicio o para comunicarse con el agente pedagógico), sino las que *no* hace, como permanecer inactivo durante mucho tiempo, paseando por el entorno pero sin llegar a hacer nada útil.

Los tres tipos de datos son conocidos en el módulo que almacena la vista lógica del mundo, gracias a la comunicación que ésta tiene con el entorno virtual y que describíamos en la sección 5.6. Con este razonamiento, pues, el módulo de percepción del agente pedagógico se puede situar directamente en la vista lógica, canalizando la información relevante que llega al módulo hacia el subsistema de tutoría. Puede verse un esquema en la figura 5.6.

Por último, el módulo cognitivo dispone de un conjunto de acciones primitivas que emite desde el subsistema de tutoría para que el avatar en el entorno actúe. Esas acciones primitivas no tienen por qué coincidir con las que veíamos que se utilizan para comunicar a la vista lógica con las entidades (ver tabla 5.1), sino que pueden ser independientes del entorno virtual utilizado y venir especificadas utilizando un vocabulario del dominio educativo particular. Por ejemplo, las acciones pueden indicar qué tipo de operaciones hay que ejecutar *sobre el simulador* de la vista lógica. El módulo encargado de la inteligencia de alto nivel del avatar, deberá traducir esa orden a acciones primitivas sobre el entorno virtual, que operen con los elementos visuales equivalentes y que, al ser ejecutadas sobre él, terminen modificando

el simulador tal y como haya solicitado el módulo cognitivo.

5.11. Relación entre entorno virtual y ejercicios

Nuestra metodología de creación de contenidos detallada en la sección 4.4 se basa en la *separación* entre los dos tipos de datos fundamentales en una aplicación educativa basada en videojuegos: aquellos relacionados con el conocimiento del dominio que se pretende enseñar, y aquellos relacionados con la parte lúdica, como son la estructura del entorno virtual, colocación de entidades o modelos tridimensionales.

Gracias a esta separación, el experto del dominio puede dedicarse a generar los ficheros que contienen los ejercicios o episodios educativos a los que el estudiante se enfrentará, mientras que el diseñador de niveles se dedicará a construir el entorno virtual por el que se moverá el usuario, y colocará las entidades clave dentro de él. Por ejemplo, en el juego educativo para entrenar bomberos, el experto del dominio puede decidir que en un ejercicio debe aparecer un edificio con un número de plantas determinado, con el fuego situado en una planta concreta y con unas características dadas. Por su lado, el diseñador de los niveles, utilizando un editor de entornos virtuales, define la arquitectura de toda la calle, coloca los elementos urbanos, y decide en qué lugar irá el edificio que tendrá fuego. Cuando el ejercicio se carga, se configurarán las propiedades del edificio, y se colocará el fuego donde proceda.

Para que esta separación sea posible, la arquitectura subyacente debe ser capaz de soportar esta configuración en función del episodio educativo que hay entre manos.

El proceso desde que un usuario entra en el sistema hasta que la aplicación está completamente lanzada es el siguiente:

- El subsistema de tutoría, concretamente el módulo pedagógico, decide las características deseadas del siguiente ejercicio.
- El gestor de ejercicios selecciona aquél que más se aproxime al solicitado. Si, como puede ocurrir, el gestor es capaz de aplicar técnicas de *adaptación*, es posible que el ejercicio seleccionado se cree *en ese momento* a partir de otros ejercicios disponibles en la base de ejercicios.
- El cambio en el ejercicio actual provoca un cambio en el estado de la lógica del mundo, especialmente en el *simulador*.
- El motor de la aplicación carga el mapa que contiene la descripción del entorno virtual. Si el mapa es muy configurable, podremos tener un mapa único, independiente del ejercicio, y por lo tanto el motor de la aplicación siempre cargará el mismo. Eso ocurre, por ejemplo, en

JV^{2}M, descrito en el capítulo 6. Si el mapa no es lo suficientemente configurable (siguiendo con el ejemplo de los bomberos, podríamos tener ejercicios con incendios en uno o dos edificios, y el diseñador del nivel ha optado por crear dos mapas independientes), deberá existir una relación entre los ejercicios y los mapas utilizados en el entorno.

- El motor de la aplicación, al *instanciar* el mapa cargado desde disco, va llamando a las funciones de creación de objetos del entorno en el módulo inmediatamente superior, según lo hemos descrito en la sección 5.8.2.

- Esas funciones llamadas por cada entidad crean los objetos del juego. Con la arquitectura de componentes, la creación de estos objetos provocará la construcción de todos los componentes que lo forman. De entre todas las posibles entidades nos interesan las siguientes:

  - Aquellas que influyen en el episodio educativo: son las que representan una parte del simulador que aparece en la vista lógica. Como vimos en la figura 5.4a, estas entidades tendrán un componente específico que se encarga de la comunicación con su representante en el simulador. En el momento de la construcción de ese componente, él mismo se encargará de, en base al estado del simulador en ese momento, establecer su apariencia inicial (recordemos que el estado del simulador ha sido configurado a partir del ejercicio en un paso previo).
  - Aquellas cuyo comportamiento de alto nivel está controlado por la vista lógica: en ese caso, el componente específico (ver figura 5.4b), emitirá la orden a la vista lógica para que cree el controlador concreto, y se da a conocer, de tal forma que en lo sucesivo puedan comunicarse para informar sobre la percepción o sobre las acciones a ejecutar en el entorno virtual.
  - Aquella que define el punto inicial del estudiante en el mapa: en ese caso, la entidad que se crea es la que representa al estudiante (figura 5.4c). El componente específico de esta entidad se encarga, igual que en el caso anterior, de informar a la vista lógica de su existencia para comenzar así la comunicación.

Al final del proceso, por lo tanto, todas las entidades del entorno virtual estarán creadas y configuradas en base a los datos contenidos en el mapa, y en base al estado del simulador.

Conviene, antes de terminar, matizar cómo debe el motor de la aplicación averiguar qué mapa debe cargar e *instanciar* dependiendo del ejercicio actual. Hemos dicho que, idealmente, el entorno virtual es lo suficientemente

configurable como para que todos los ejercicios puedan ejecutarse a partir del mismo mapa inicial. En JV^{2}M por ejemplo, esto se cumple.

En caso de no ser posible crear un mapa lo suficientemente configurable, debe implementarse algún mecanismo de *emparejamiento* que, dado un ejercicio, indique qué mapa hay que cargar. Para intentar resolver el problema de la manera menos intrusiva posible, ese emparejamiento *no* debería hacerse en el propio ejercicio. Es decir, el problema de la configuración no debe recaer sobre el autor del ejercicio. Lo que marca qué mapa hay que cargar son las propias *propiedades* del ejercicio (por ejemplo, el número de edificios incendiados), por lo que la relación tiene que realizarse en base a esas características. La solución es implementar esa decisión en un módulo aparte que conoce las causas de la necesidad de varios mapas, y decide de acuerdo a ellas, a partir de las propiedades del ejercicio seleccionado.

5.12. Evaluación

La arquitectura que hemos presentado en este capítulo no es el resultado de un mero diseño en papel, sino que se ha llevado a la práctica en distintas instanciaciones, que prueban su versatilidad.

Los ejes de variabilidad que hemos probado han sido los siguientes (ver tabla 5.2):

- Cambio del motor de la aplicación:
  - En un prototipo inicial, el motor gráfico utilizado fue WildMagic[1], sobre OpenGL y utilizando GLUT como librería para gestionar la entrada. Los mapas no eran leídos desde disco, sino que estaban programados directamente en el código; los modelos animados se cargaban de ficheros con el formato usado en el juego Quake 3.
  - En otra instanciación de la arquitectura, utilizamos Nebula 1, un motor de juegos que se encarga de la gestión de entrada, el dibujado tanto tridimensional como bidimensional, y otra serie de características. La carga de los mapas se hacía desde ficheros `bsp`, formato utilizado en el juego Half-Life 1, y los modelos animados también utilizaban el formato de este juego.
  - Por último, cambiamos a Nebula 2, una versión renovada del motor anterior. Los cambios de funcionamiento que tiene con respecto a la versión previa son lo suficientemente significativos como para que la conversión exitosa pueda entenderse como un logro de la arquitectura. Los mapas se cargan o bien desde un `bsp` o desde un XML (ver apartado 6.7.2 para más detalles), y los modelos animados utilizan el formato propietario del propio motor gráfico.

[1] `http://www.geometrictools.com/`

	Prototipo	JV^{2}M 1	JV^{2}M 2	VirPlay
Motor de la aplicación	WildMagic + GLUT	Nebula 1 / Nebula 2	Nebula 2	Nebula 1
Mecánicas	Aventura gráfica	Aventura gráfica	Acción	Aventura gráfica
Contexto educativo	Ninguno	JVM	JVM	Patrones de diseño
Mapas	Cableado	`bsp` (Half-Life 1)	`bsp` / XML	`bsp` (Half-Life 1)
Modelos animados	`md3` (Quake 3)	`mdl` (Half-Life 1)	`n2` (Nebula 2)	`mdl` (Half-Life 1)

Tabla 5.2: Distintas instanciaciones de la arquitectura

- Cambio de la temática del juego: como veremos en el capítulo siguiente, el proyecto JV^{2}M tiene dos versiones, cuyo fondo es el mismo (la enseñanza de la máquina virtual de Java, y el proceso de compilación), pero cuyas mecánicas en cuanto a juego son distintas:
  - La primera versión está claramente inspirada en las aventuras gráficas, y por lo tanto, las acciones que tanto el estudiante como el agente pedagógico que habita el entorno hacen son las de "mirar", "coger", "usar" y "usar con", descritas en la sección 5.6.
  - La segunda versión es similar a un juego de acción, donde el estudiante lleva un arma, y tiene que eliminar enemigos para hacerse con los objetos necesarios para terminar el ejercicio.
- Cambio de dominio: como ya hemos dicho, la arquitectura la hemos utilizado en JV^{2}M, una aplicación educativa para enseñar la máquina virtual de Java. Además, también ha sido utilizada en el desarrollo del primer prototipo de VirPlay (Jiménez Díaz, Gómez Albarrán, Gómez Martín y González Calero, 2005b), una aplicación educativa que, utilizando *role play*, enseña la interacción entre los distintos objetos que intervienen en distintos patrones de diseño.

Resumen

En este capítulo hemos presentado la arquitectura para el desarrollo de aplicaciones educativas basadas en videojuegos. Tras una introducción, la sección 5.2 detalla los objetivos que guiaron el diseño de la arquitectura.

Posteriormente, la sección 5.3 hace una descripción general, enumerando los módulos que la forman. A continuación, las distintas secciones van de-

tallando cada uno de esos módulos. Posteriormente se dedica un apartado al control de la ejecución y la incorporación de agentes pedagógicos en las aplicaciones.

Una vez descrita por completo la arquitectura, el capítulo recupera el aspecto más importante de la metodología descrita en el capítulo anterior. En particular, se indica el modo de unir los dos tipos de información fundamentales, a saber, ejercicios y entornos virtuales.

Por último, la sección 5.12 presenta una evaluación, mostrando brevemente las distintas instanciaciones de la arquitectura que hemos implementado en forma de distintas aplicaciones educativas. En particular, se ha indicado qué módulos han sido reemplazados de una implementación a otra.

El capítulo siguiente hará una descripción extensa del uso de la arquitectura en JV^{2}M. En particular, describiremos con detalle la implementación concreta de cada uno de los módulos.

Notas bibliográficas

Las decisiones arquitectónicas que hemos tomado en este capítulo se han fundamentado en lo visto en los capítulos 2 y 3, así como en los objetivos marcados en la sección 5.2.

Una versión preliminar de la arquitectura fue presentada en Gómez Martín et al. (2003). Para una descripción extendida, se puede consultar Gómez Martín, Gómez Martín y González Calero (2007a) y Gómez Martín, Gómez Martín y González Calero (2007b). Por su parte, Jiménez Díaz, Gómez Albarrán, Gómez Martín y González Calero (2005c) describe en detalle la aplicación VirPlay y da una explicación de cómo se ha utilizado la arquitectura propuesta en este capítulo en su implementación.

Capítulo 6

Sistema educativo para la enseñanza de la máquina virtual de Java

Optó por la memorización mecánica, aunque sabía que, en el mejor de los casos, sólo le brindaría una educación hueca.

Carl Sagan. Contact

En este capítulo se describen las dos versiones JV^{2}M, el sistema educativo que permite aprender la máquina virtual de Java y cómo el código Java es compilado para ella. Este sistema ha sido creado utilizando la metodología y arquitectura descritas en los dos capítulos anteriores.

6.1. Introducción

El capítulo anterior está dedicado a la arquitectura general propuesta para la creación de un sistema educativo basado en videojuegos. El máximo exponente de esta arquitectura es JV^{2}M, un sistema para la enseñanza de la máquina virtual y la compilación de Java.

En este capítulo nos centramos en este sistema, detallando la forma en que han sido instanciados los módulos más importantes descritos de manera general en el capítulo anterior. También prestaremos atención a la parte metodológica descrita en el capítulo 4, en particular cómo realizamos la creación de contenidos.

Como quedaba reflejada en la sección de evaluación del capítulo anterior (sección 5.12), JV^{2}M ha ido pasando por distintas versiones, que han ido

provocando la sustitución de los distintos módulos afectados. En la siguiente sección se describen las metáforas utilizadas para la representación de la JVM en las dos versiones principales. Para su total comprensión, es necesario al menos un conocimiento superficial de la estructura de la JVM, por lo que el apéndice A realiza una introducción a sus componentes principales.

Una vez descritos ambos sistemas, el capítulo detalla cada uno de los módulos que proponía la arquitectura para la construcción de sistemas educativos basados en videojuegos (capítulo 5). En particular, visita cada uno de los módulos, indicando de qué forma se han implementado en JV^{2}M.

El capítulo termina indicando de qué forma se han creado los contenidos que utiliza JV^{2}M. Como se podrá comprobar el método utilizado es el propuesto en la metodología de creación de contenidos del capítulo 4.

6.2. Descripción general

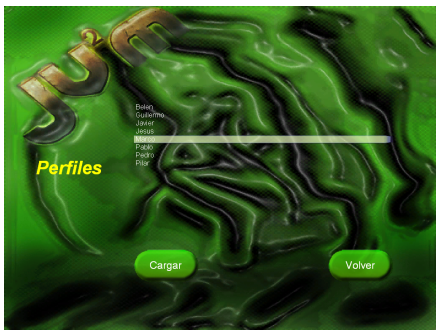
Nuestro objetivo es implementar un sistema educativo basado en videojuegos para enseñar la estructura de la máquina virtual de Java (JVM) y la forma en la que el código fuente desarrollado en Java es compilado para generar las instrucciones que ella entiende. Para ello, suponemos que los usuarios del sistema conocen la programación imperativa en general y Java en particular. Con el sistema serán capaces de aumentar su conocimiento en programación orientada a objetos y el proceso de compilación para una máquina de tipo pila como la JVM. El sistema, no obstante, no cubre todos los conceptos de la máquina virtual y del lenguaje Java. En la sección 6.2.3 detallaremos exactamente cuáles de los aspectos no son cubiertos.

El sistema presenta un entorno virtual que simula la JVM, por lo que ha sido bautizado como JV^{2}M, para indicar la idea de una máquina virtual *virtualizada*.

En ese entorno virtual, el estudiante es representado mediante un avatar con el cual interactúa con el resto de objetos. El tipo de objetos, utilidad y significado concreto es dependiente del diseño del juego utilizado. Como explicábamos en la sección 4.2.1, antes de abordar la implementación de un videojuego educativo se debe pasar por una fase de *reflexión* en la que se decide, además del ámbito que se desea enseñar, las características del juego que van a servir como método de aprendizaje. En concreto, se debe determinar el tipo de juego, mecánicas que tendrá, acciones del usuario, etc.

JV^{2}M ha pasado por dos versiones principales, JV^{2}M 1 y JV^{2}M 2, con diseños muy distintos. La primera versión se basa en las mecánicas de una aventura gráfica al estilo de la saga de Monkey Island o Grim Fandango (LucasArts, 1998), mientras que la segunda es un juego de acción.

En ambos casos, no obstante, los episodios de aprendizaje consisten en enfrentar al alumno con un programa realizado en Java que debe *ejecutar* en



(a) Pantalla de selección de usuario en JV^{2}M 2

(b) Pantalla de menú de JV^{2}M 2 donde se muestra el usuario activo

Figura 6.1: Capturas de JV^{2}M 2 relacionadas con la selección de usuario

la máquina virtual simulada. Para ello, tendrá que *compilar* el código Java a instrucciones de la máquina virtual (*bytecodes*), y hacer que se ejecuten cada una de ellas en el entorno de acuerdo con la metáfora.

Las dos versiones de JV^{2}M ejecutan el ciclo de aprender-haciendo (Gómez Martín, Gómez Martín y González Calero, 2005a). Para eso, en primer lugar, el usuario se identifica en el sistema (figura 6.1). Una vez seleccionado el perfil, el sistema elige el ejercicio más conveniente para su nivel de maestría, y se le enfrenta a él en el entorno virtual.

Desde el punto de vista *externo*, las diferencias fundamentales entre ambas versiones de JV^{2}M radican en el diseño o mecánicas del juego (Gómez Martín, Gómez Martín, González Calero y Palmier Campos, 2007d).

La primera versión (JV^{2}M 1) está basada en las aventuras gráficas, especialmente aquellas que disponen de escenarios en tres dimensiones realizadas por LucasArts, como Escape from Monkey Island o Grim Fandango.

Por su parte, la segunda versión (JV^{2}M 2) utiliza mecánicas de juegos de acción; el estudiante/jugador está equipado con un arma, y debe eliminar a otros personajes que hay en el entorno para conseguir recursos que le permitan ejecutar el código Java.

En las dos secciones siguientes describimos cada una de las dos versiones. Posteriormente, detallamos qué características concretas de la JVM no son cubiertas por ninguna de nuestras aplicaciones educativas.

6.2.1. Descripción de JV^{2}M 1

En ambas versiones prácticamente todos los componentes de la JVM están representados por objetos en el entorno virtual, aunque dado que la ambientación y modo de interacción es distinta, la forma de representación difiere.

(a) Jugador enfrente del edificio que representa el *frame* activo

(b) Código máquina y Java en el interior del *frame*

(c) Frame hundiéndose en la pila, para dejar hueco para la llegada del siguiente

(d) Nuevo *frame* apilándose en la pila de frames

Figura 6.2: Distintas capturas de JV^{2}M 1 relacionadas con la pila de frames

En la primera versión, el usuario se pasea por un entorno exterior donde aparecen algunos edificios a los que también puede entrar. La forma de representar las distintas estructuras de la JVM (listadas en la sección A.2) es la siguiente:

- **Heap**: el conjunto de objetos creados en la máquina virtual es representado mediante un *barrio* de edificios, cada uno de ellos simbolizando uno de los objetos creados. El estudiante/jugador puede entrar en cada edificio para poder acceder a los valores de los atributos del objeto al que representa.
- **Área de métodos**: en vez de tener un espacio físico donde se encuentra el código de todos los métodos, el entorno virtual dispone de otro *barrio*, esta vez de clases, donde cada edificio representa una clase cargada en la máquina virtual. Cada uno de los edificios representa a una de ellas, de tal forma que cuando el estudiante entra en el edificio de una clase, puede acceder al código de sus métodos de esa clase y al valor de sus

atributos estáticos.

- **Pila de frames**: en el entorno aparece únicamente un edificio que representa el *frame* activo en ese momento (figura 6.2a). Dentro de él, se puede consultar el código del método (figura 6.2b), ver sus variables locales y manipular la pila de operandos. Para las invocaciones a métodos nuevos (o terminar la ejecución del método activo), el estudiante debe salir fuera del edificio. Al crear un *frame* nuevo, el edificio desaparece bajo tierra (figura 6.2c), y cae uno nuevo, simulando la acción de *apilar* un nuevo registro de activación en la pila (figura 6.2d). Cuando un método termina, se realiza el proceso inverso.

- **Área de constantes**: en nuestra metáfora, no existe un área de constantes explícitamente, sino que las referencias son *resueltas* en el momento de inicialización del ejercicio, de tal forma que, cuando se muestra el código, se ven los operandos *reales* (y no las referencias), y en caso de que el usuario las necesite, aparecen automáticamente en su inventario.

- **Contador de programa**: igual que en el caso del área de constantes, *no* existe una representación metafórica como tal del contador de programa. El jugador/estudiante es el único que conoce en realidad cuál es la instrucción que se está ejecutando. En los primeros ejercicios, donde aparece tanto el código Java como el código compilado, la instrucción máquina que hay que ejecutar aparece remarcada (como puede verse en la figura 6.2b), pero no consideramos eso como contador del programa, ya que el usuario no puede interactuar con él.

Dependiendo del tipo de instrucción máquina de la JVM, los pasos que tiene que realizar el estudiante en el entorno difieren:

- **Instrucciones de carga, almacenamiento y manipulación de la pila**: son las instrucciones que copian datos desde el área de variables locales de un *frame* a su pila de operandos y viceversa. Independientemente de la dirección, todas las instrucciones tienen un parámetro que indica el *número* de variable local implicada. El estudiante interactúa moviendo las cajas que representan los valores.

- **Instrucciones aritméticas y de conversión de tipos**: los cálculos finales de estas operaciones son ejecutados por un personaje en el entorno que juega el papel de unidad aritmético-lógica (ALU); el personaje aparece parcialmente en la figura 6.2b, entre el código Java y los *bytecodes*. El estudiante interactúa con él para solicitarle que realice las distintas operaciones.

(a) Estudiante y Javy dirigiéndose al barrio de clases

(b) Estudiante enfrente de Framauro

(c) Estudiante interactuando con la pila de operandos

(d) Estudiante hablando con Javy

Figura 6.3: Capturas de JV^{2}M 1

- **Creación y acceso a objetos**: para la creación de objetos (edificios en el barrio que representa al *heap*), se solicita la creación a otro avatar del entorno. Una vez creado, la manipulación de sus atributos se realiza en el interior de ese edificio. Si los atributos son estáticos, se debe operar dentro del edificio que representa a la propia clase.
- **Instrucciones de salto**: dado que es el propio estudiante el que está ejecutando el código objeto, es él mismo el que hace las veces de contador de programa. Por lo tanto, las operaciones de salto no se ejecutan *explícitamente*, sino que simplemente la porción de código afectada se repite o se deja de ejecutar.
- **Invocación de métodos**: es el propio estudiante el que debe ejecutar el mecanismo de *resolución* de los métodos, buscando el método en la clase a la que pertenece el objeto que recibe el mensaje, y efectuando la búsqueda entre las clases padre hasta que se encuentra. Una vez encontrado el método real a ejecutar, y con una referencia a él, se confía en un avatar del entorno llamado Framauro (figura 6.3b) que

está situado a la entrada del *frame* activo, para que construya el nuevo *frame*.

Para la ejecución de todas las instrucciones, el estudiante cuenta con la ayuda de JAVY (ver figura 6.3d), un agente pedagógico con el que el usuario puede entablar una conversación, y a quién, en última instancia, puede solicitar que termine de resolver el ejercicio.

La forma en la que el usuario interactúa en JV^{2}M 1 con el entorno (esto incluye al propio JAVY), se basa, como ya se ha comentado, en las aventuras gráficas. De esta forma, el usuario puede realizar una serie de operaciones sobre los objetos del entorno (ver el apartado 6.5 para más detalles).

La parte más destacada de la manipulación es que la ejecución de una única instrucción de la JVM puede requerir interactuar con varios objetos del entorno, cada uno representando una estructura de la JVM diferente. Cada uno de los pasos u operaciones que hay que realizar sobre el entorno los hemos llamado *microinstrucciones*. Cuando el estudiante pide ayuda a JAVY, este puede indicarle qué *microinstrucción* debe ejecutar a continuación, o ejecutarla por él mismo.

6.2.2. Descripción de JV^{2}M 2

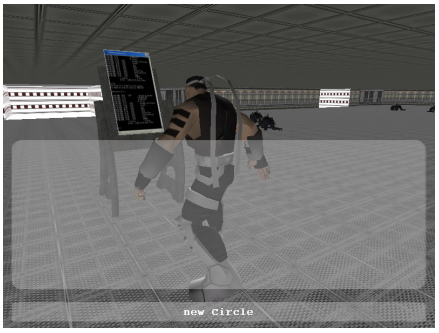
El prototipo de JV^{2}M 1 si bien vino a demostrar que la arquitectura y metodología descrita en los capítulos 4 y 5 es factible, también reflejó que la ejecución de una única instrucción máquina podía llevar demasiado tiempo, ya que exigía al estudiante interactuar con varios objetos distintos.

Para aliviar este problema, optamos por cambiar por completo el diseño del juego, y abordar la implementación de otra aplicación educativa distinta con la misma arquitectura y objetivos pedagógicos.

El nuevo diseño se basa en juegos de acción. El entorno en el que el estudiante se mueve es una nave espacial que representa, de forma metafórica, el estado de la máquina virtual, y donde también existen otros personajes a los que el estudiante tiene que eliminar.

Desde el punto de vista pedagógico, el mecanismo básico es el mismo que en la versión anterior. El usuario es enfrentado a un código Java que debe compilar y ejecutar. La diferencia fundamental a este respecto es que en JV^{2}M 1 el estudiante debe ejecutar cada una de las instrucciones ejecutando *microinstrucciones* sobre el entorno, mientras que en esta nueva versión la ejecución consiste simplemente en teclear en un terminal las instrucciones máquina (ver figura 6.4a).

Lo que sí se mantiene igual es que los valores que se almacenan en la JVM (tanto en la pila de operandos, variables locales como atributos) están representados por cajas. Mediante una visión de rayos X, el estudiante puede comprobar el valor de cada una de esas cajas (figura 6.4b).



(a) Estudiante utilizando el terminal para ejecutar instrucciones

(b) Estudiante comprobando el contenido de una caja de la pila de operandos

Figura 6.4: Capturas de JV^{2}M 2

La forma de representar en esta nueva metáfora las estructuras de la JVM listadas en la sección A.2.2 es:

- **Heap**: representado por una sala de la nave espacial, todos los atributos de un objeto particular se encuentran agrupados en un conjunto de cajas. El estudiante puede almacenar una copia de algunos de esos datos en el inventario, o modificar su contenido.
- **Área de métodos**: igual que en JV^{2}M 1, el área de métodos se encuentra distribuida por clases. Existe una habitación por cada clase cargada en la JVM, colocadas de tal forma que es fácil identificar la jerarquía entre ellas. Dentro de la clase un terminal da acceso al código Java de todos los métodos que implementa esa clase.
- **Pila de frames**: en este caso, cada uno de los *frames* consiste en un *piso* o *nivel* distinto del entorno. Para crear un nuevo registro de activación, hay que utilizar el ascensor que da acceso a él. Para bajar a un método cuya ejecución se ha interrumpido, simplemente se baja de piso.
- **Contador de programa**: igual que en JV^{2}M 1, no existe un representante virtual del contador de programa, ya que es el propio usuario el que sabe por qué instrucción va.
- **Área de constantes**: las constantes que en JV^{2}M 1 aparecían automáticamente en el inventario del jugador, ahora están representadas por objetos del entorno que hay que recolectar para poder utilizarlos en las instrucciones tecleadas en el terminal.

Con esta metáfora, la mayoría de las instrucciones máquina se ejecutan directamente tecleándolas desde el terminal (figura 6.4a), una vez que se dispone de los recursos (argumentos) necesarios.

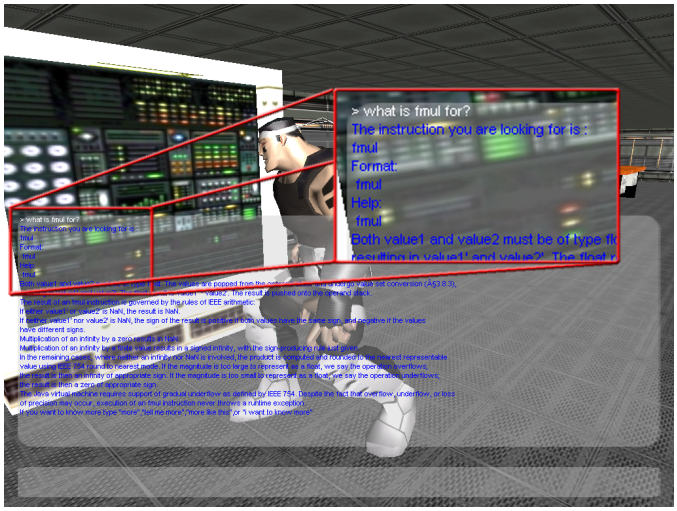


Figura 6.5: Ayuda en JV^{2}M 2

Javy, el agente pedagógico de la versión anterior ha sido sustituido por una ayuda semántica que permite al usuario desde el terminal lanzar preguntas al sistema (figura 6.5). El funcionamiento de esa ayuda semántica queda fuera del alcance de este trabajo, y es descrito por Gómez Martín (2007).

La principal diferencia entre las metáforas de JV^{2}M 1 y JV^{2}M 2 es la forma en la que se ejecutan las instrucciones. Mientras que en el primero se trata de la manipulación de objetos, en la segunda prácticamente todas las instrucciones son ejecutadas mediante el terminal. Si el usuario intenta ejecutar una instrucción para la que aún no ha conseguido los recursos (parámetros) necesarios, el terminal avisa del error.

Como resumen de las diferencias entre las metáforas de las dos versiones de JV^{2}M, la tabla 6.1 presenta una comparación entre la forma de representar en ellas las distintas estructuras de la JVM.

6.2.3. Aspectos de la JVM no cubiertos

Las dos aplicaciones educativas sirven para enseñar el funcionamiento interno de la JVM, así como la forma en la que el código Java se compila a código objeto o lo que es lo mismo, a los *bytecodes* directamente interpretables por la máquina virtual.

Sin embargo, ninguna de las dos versiones cubre todos los aspectos de la máquina virtual indicados en el apéndice A. Las simplificaciones más importantes llevadas a cabo son las siguientes:

- La JVM permite varias hebras en ejecución simultáneamente, duplican-

Metáfora		
JVM	JV^{2}M 1	JV^{2}M 2
Clases (código de métodos)	Barrio de clases	Habitaciones
Heap	Barrio de objetos	Habitaciones
Frame	Edificio	Nivel de la nave
Pila de operandos	Cajas en una mesa	Cajas
Variables locales	Armario	Cajas
Tabla de constantes	Inventario	Avatares a capturar
Instrucciones máquina		
Manipulación de la pila	Manipulación de cajas	Terminal
Aritméticas y conversión de tipos	Avatar (ALU)	Terminal
Creación de objetos	Avatar	Terminal
Salto (*invokevirtual*)	Framauro	Ascensor

Tabla 6.1: Resumen de las metáforas de JV^{2}M 1 y JV^{2}M 2

do la estructura de la pila de registros de activación. Nuestra aplicación educativa sólo permite una hebra.

- En ambas metáforas disponemos de una zona donde aparecen todas las clases cargadas. En realidad, se ocultan o simplifican muchos detalles que la JVM permite. En particular, el estudiante no es responsable de asegurar que las clases que se utilizan estén ya cargadas. Tampoco se permite distintos cargadores de clases, algo que sí es factible en la JVM para conseguir distintos *universos de clases* independientes entre sí para aumentar la seguridad.

- La JVM dispone de excepciones a dos niveles: excepciones que lanza el programador, y excepciones de la propia máquina virtual, lanzadas o bien por errores graves del programa (clase no encontrada, división por cero, acceso inválido a un array) o bien de la propia máquina virtual (falta memoria). Ninguno de estos tipos es tratado por nuestra aplicación.

- La máquina virtual de Java es capaz de ejecutar las instrucciones listadas en el apéndice A. Sin embargo, esas instrucciones *no* son suficientes para implementar cualquier programa (sin ir más lejos, no hay instrucciones relacionadas con la entrada/salida). El resto de funcionalidad se consigue a base de la implementación de métodos en código nativo. Son métodos de clases que, en vez de estar implementados en Java, están implementados en otro lenguaje distinto (normalmente C/C++) y que utilizan los recursos del sistema operativo subyacente para realizar su función. Nuestra aplicación educativa *no* permite salirse del código

Java, por lo que los métodos nativos y todo lo que ello implica (pila de ejecución para estos métodos, implementación del interfaz con el código nativo o JNI) no son tratados.

- Dada la restricción anterior, la máquina virtual en la que el estudiante ejecuta las instrucciones, tampoco dispone del conjunto de clases de la librería de Java, pues una gran parte de ellas (las de más bajo nivel) terminan haciendo uso de los métodos nativos.

- Por último, y relacionado con el hecho de no disponer de la librería de Java, surge el caso especial de la clase `java.lang.Object`. Ésta es una clase de especial importancia tanto en la librería como en el propio lenguaje Java y la JVM, ya que todas las clases heredan automáticamente de ella. Dado que nuestra implementación olvida la librería de Java completa, tampoco tiene esta clase. Eso es debido a que la clase `Object` tiene una serie de métodos que abren la puerta a otras nuevas clases que obligarían a incluir muchas clases donde se mezcla la implementación de la máquina virtual, los métodos nativos y la librería de Java[1]. Sin embargo, desde el punto de vista pedagógico, no nos podemos permitir el lujo de eliminar la clase por completo, ya que es una parte demasiado importante del lenguaje Java. Para disponer de la clase `Object` pero no necesitar la implementación de todos sus métodos, exigimos a todos los ejercicios que dispongan de una clase *artificial* de la que heredan el resto de clases, y que viene a jugar el papel de `java.lang.Object`.

6.3. Motor de la aplicación

Ya dijimos en la sección 5.8 del capítulo anterior que éste es el módulo de más bajo nivel de toda la aplicación, sobre la que se apoyan el resto de sistemas.

Tiene subsistemas básicos, como el gestor de memoria, de ficheros, funciones matemáticas y gestión de mensajes de depuración/log. El resto de la aplicación puede utilizar estos sistemas para facilitar la implementación. Como dijimos en la sección 5.8, es aconsejable que al menos el resto de los subsistemas del propio motor de la aplicación y el gestor de entidades los usen.

La arquitectura es capaz de soportar un cambio completo del motor de la aplicación, de tal forma que se puedan utilizar motores de juego cuyo interfaz está documentado, como por ejemplo el de Half-Life o Far Cry.

[1] Por ejemplo el método `getClass` devuelve un `java.lang.Class`, punto de entrada al sistema de información de tipos en tiempo de ejecución.

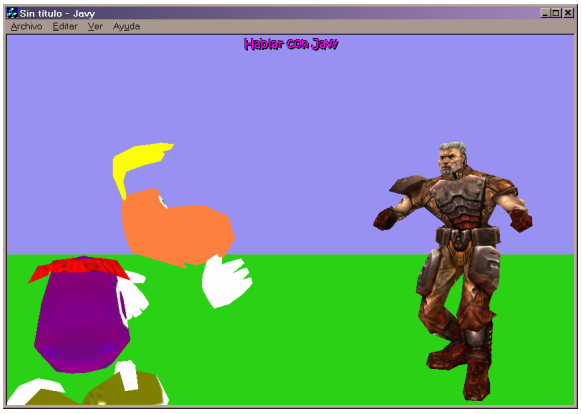


Figura 6.6: Captura de JV^{2}M 1 preliminar, utilizando WildMagic

Para la implementación de JV^{2}M, no obstante, hemos utilizado un motor de aplicación propio, formado por distintos módulos, cada uno de ellos realizados desde cero, o basándonos en otras librerías o motores. A continuación aparece la descripción de la implementación de los más importantes.

6.3.1. Motor gráfico

Se encarga de la parte de dibujado del entorno virtual, así como de la creación de la ventana donde se muestra todo el interfaz de usuario.

En nuestro caso, la evolución de JV^{2}M nos ha llevado a utilizar hasta tres motores gráficos distintos, que describimos a continuación:

- **WildMagic**: fue el primer motor gráfico que utilizamos. Está desarrollado en C++ y se distribuye bajo licencia LGPL[2]. Su documentación viene en forma de libros publicados por el programador principal (Eberly, 2000, 2004), por lo que en general es fácil su uso, gracias a la extensa documentación. La figura 6.6 muestra una captura temprana de JV^{2}M 1 utilizando éste motor.

- **Nebula 1**: este motor es producto de la política de una empresa alemana[3] que decidió liberar el código del motor que utilizaban en sus juegos comerciales, con el fin de que éste mejorara gracias a las contribuciones externas. La principal ventaja con respecto al motor anterior y que motivó la migración es el número de capacidades ofrecidas no disponibles en WildMagic. Nebula no es un simple motor gráfico, sino que incluye también otros subsistemas nombrados en la arquitectura,

[2] `http://www.geometrictools.com/`
[3] La empresa es RadonLabs, `http://www.radonlabs.de/`.

Figura 6.7: Modelo de Half-Life cargado con Nebula 1

como un gestor de ficheros, memoria y log o control de la entrada. Además, dispone de soporte nativo para lenguajes de scripts como TCL, lo que permite el desarrollo rápido de aplicaciones sencillas (Gómez Martín y Gómez Martín, 2005).

- **Nebula 2**: es la evolución del motor gráfico anterior. El código fuente también está disponible y se distribuye bajo una licencia que, igual que en el caso anterior, permite su uso por terceros sin coste adicional. A pesar de compartir el nombre con la versión anterior, los cambios entre ambas versiones son notables[4], principalmente en el motor gráfico debido al soporte de los *pipeline* gráficos programables (*shaders*). Una comparación entre ambas versiones puede extraerse de Gómez Martín y Gómez Martín (2006).

Para minimizar el impacto provocado por los cambios en el motor gráfico, se ha utilizado, como veremos a continuación, una *fachada* formada por un conjunto de clases independientes del motor utilizado. Esta fachada oculta al resto de la aplicación (y con esto no solo nos referimos al código específico[5], sino también al resto del motor de la aplicación) las peculiaridades concretas del mismo.

Esa fachada, no obstante, únicamente supone una barrera en cuanto al código, pero, lamentablemente, no en cuanto a los datos. El motor gráfico impone unas restricciones sobre los modelos que es capaz de dibujar, y por consiguiente, también respecto a los formatos de los ficheros que los contienen. Puede ocurrir incluso que el motor no sea capaz de leer ningún tipo de

[4] `http://nebuladevice.cubik.org/what-is-n2/features/`

[5] En este apartado del motor de la aplicación recuperaremos el significado original de "código específico" que veíamos en la figura 4.2, es decir "todo lo que no es motor de la aplicación".

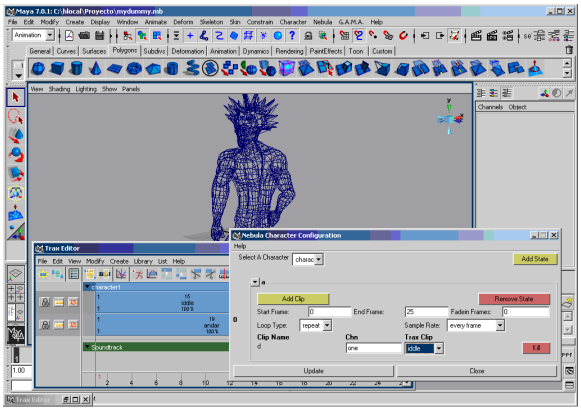


Figura 6.8: Creación de un modelo para Nebula 2

fichero concreto; en ese caso, se debe ampliar el motor para que sea capaz de interpretar alguno.

Para poder presentar personajes con WildMagic, extendimos el motor para soportar modelos de Quake 3 (ver figura 6.6), un juego que disponía de personajes animados por *keyframes*; Nebula 1 fue extendido para dibujar modelos de Half Life (figura 6.7) cuyas animaciones utilizaban huesos, ahorrando así memoria durante la ejecución; por último, en Nebula 2 utilizamos el formato del propio motor gráfico, ya que existe un plug-in comercial que permite grabar los modelos creados en Maya a ficheros directamente interpretables por él (figura 6.8). Veremos en la sección 6.7.2 que la existencia de estos plug-in exportadores de modelos permiten mantener independiente el trabajo de los diseñadores del motor final utilizado; en nuestro caso, nos permitió utilizar en Nebula 2 algunos de los modelos creados para ser dibujados en Nebula 1.

Retomando el mecanismo de protección del *código* frente a esos cambios, hemos mencionado que utilizamos un diseño de clases que actúan como fachada. En particular, la fachada evita los siguientes cambios:

- Cambios en la inicialización/destrucción: cada uno de los motores gráficos anteriores requieren una serie de pasos distintos tanto al principio como al final de la aplicación.
- Cambios en el bucle principal: el motor de la aplicación incluye el control de la ejecución o bucle principal (que veremos en la sección 6.3.6). En cada vuelta del bucle, como ya tratamos en el apartado 5.9, se debe invocar al motor gráfico, para que dibuje el fotograma correspondiente, teniendo en cuenta todas las entidades registradas. La forma de invocación concreta es dependiente del motor gráfico, por lo que una fachada que lo oculte evita el cambio en el bucle principal.

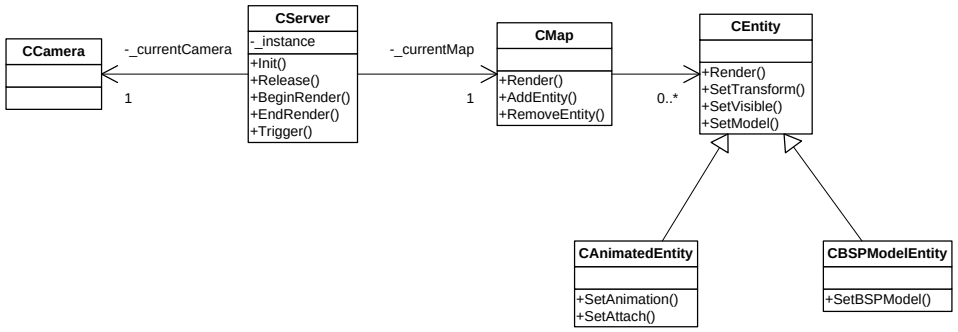


Figura 6.9: Clases públicas del motor gráfico de JV^{2}M

- Cambios en el código específico: los objetos de juego con representación gráfica, deben invocar al motor gráfico para que cargue los modelos concretos. La forma de invocar al motor varía entre uno y otro.

El diseño de clases que se publicita al resto de la aplicación desde el motor gráfico consta, principalmente, de las seis clases mostradas en la figura 6.9[6].

La más importante, `CServer` es un singleton con inicialización explícita. Su misión principal es almacenar todas las referencias al motor gráfico lanzado, y ocultar los detalles de su inicialización y destrucción. Tiene tres métodos que son llamados en cada vuelta del bucle principal: `Trigger` que realiza funciones de mantenimiento (como procesar los eventos de la ventana) y `BeginRender` y `EndRender` que son métodos invocados justo antes y después del dibujado. Estos dos métodos son requeridos por todos los motores gráficos, para configurar el *hardware* para el dibujado o la presentación final de la escena.

Destaca que el servidor *no* tiene un método para dibujar la escena. El dibujado concreto es responsabilidad de la clase `CEntity` y sus derivadas, y de la clase `CMap`.

La primera de ellas representa a una entidad u objeto gráfico que detallaremos a continuación. La segunda es una simple agrupación de entidades. La peculiaridad es que el motor gráfico tiene en cada momento uno de estos *mapas* activos. Desde el bucle principal de la aplicación, se invoca al dibujado de ese mapa activo que, sucesivamente, va llamando al método de dibujado de todas las entidades registradas en él. Por lo tanto, la clase `CMap` es la que permite la independencia del bucle principal con el motor gráfico: en cada vuelta del bucle, simplemente se llama al método `Render` del mapa actual, el cual llamará al mismo método de todas las entidades, cuyo código, ahora

[6]Tanto en la figura como en la explicación que sigue, se ha omitido el nombre del *namespace* utilizado; todas las clases descritas a continuación pertenecen al módulo o namespace `Graphics`.

sí, será específico del motor gráfico.

En cuanto a las entidades gráficas, éstas representan los modelos que se dibujan. La clase base, `CEntity` tiene métodos para establecer la posición y rotación, establecer su visibilidad, así como para indicar el nombre del fichero que contiene el modelo gráfico[7]. La clase, además, es extendida por otras clases específicas, que permiten características avanzadas. Por ejemplo, `CAnimatedEntity` sirve para modelos de personajes animados, permitiendo cambiar la animación o adjuntar otros modelos como armas o escudos. Por su parte, `CBSPModelEntity` permite cargar modelos gráficos desde ficheros de mapas BSP, cuya importancia quedará reflejada en la sección 6.3.3.

Para finalizar, la última clase *pública* de la fachada es la que representa la cámara utilizada, que fija la posición desde la que se está viendo la escena dibujada, y que contiene los métodos habituales de control de posición y apertura de lentes. La implementación traduce esas órdenes generales al lenguaje específico utilizado por el motor gráfico subyacente.

6.3.2. Gestión de entrada

La captura de los eventos de entrada del usuario (pulsación de teclas, movimiento de ratón y/o palancas de juego) es realizado también en el motor de la aplicación.

En nuestro caso, en la primera versión utilizamos GLUT[8] (*The OpenGL Utility Toolkit*), una librería pensada para aplicaciones que hacen uso de OpenGL para renderizar (como utilizábamos con WildMagic). La librería procesa los eventos de usuario y permite su captura por parte de la aplicación.

Posteriormente, con la sustitución de WildMagic por Nebula 1 y más tarde por Nebula 2, cambiamos el sistema de captura, para pasar a utilizar el proporcionado por Nebula. En este sentido, Nebula dispone de una *cola de eventos* que la aplicación puede procesar en el bucle principal para actuar ante ellos.

Independientemente de cómo se acceda al *hardware* para leer la entrada, el motor de la aplicación proporciona la funcionalidad necesaria para que el código específico, mediante el patrón observer (Gamma et al., 1995) sea informado de la entrada del usuario y reaccione ante ella.

En concreto, las cuatro clases más importantes aparecen en la figura 6.10. Hay dos clases que sirven de *gestores* de la entrada por teclado y ratón, y otras dos a modo de *interfaz* que deben implementar todas las clases que quieran ser informadas de los eventos, y que se registran en las primeras.

[7] Obsérvese que el fichero *sí* es dependiente del motor gráfico subyacente, al tener la dependencia de datos antes comentada. Sin embargo, el nombre del fichero utilizado vendrá a su vez de otro fichero de datos (mapa) que deberá cambiarse en caso de cambio de motor. Lo que no habrá que alterar será el código fuente de otra parte de la aplicación, garantizando la independencia con respecto al motor del resto del código.

[8] `http://www.opengl.org/resources/libraries/glut/`

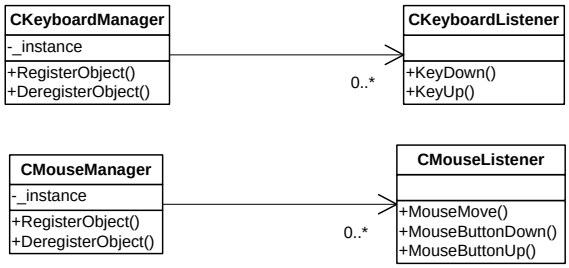


Figura 6.10: Clases relacionadas con la entrada de JV^{2}M

Esta forma de comunicación entre el motor de la aplicación y el código específico se ha mantenido desde la versión preliminar con WildMagic, de forma que el código específico no se ha visto afectado por el cambio de procedimiento de lectura. En todas las versiones el bucle principal de la aplicación detecta la entrada (de formas distintas en cada caso), y la redirige al gestor correspondiente para que el código específico sea informado.

6.3.3. Cargador de mapas

Ya dijimos en la sección 5.8.2 que es el motor de la aplicación el que se encarga de la carga del mapa o nivel, donde están todos los datos del entorno.

Este módulo impone también una fuerte restricción en cuanto a los ficheros que se van a poder utilizar para guardar los mapas. Tradicionalmente, los distintos motores de juego utilizados por los estudios utilizan sus propios ficheros de datos, generados por los editores que ellos mismos desarrollan. En nuestro caso, hemos optado por un diseño de clases que permite el soporte de distintos tipos de mapas (ver figura 6.11), y lo hemos instanciado para ficheros `bsp` utilizados en juegos como Half-Life o Quake y para ficheros cuyos datos están en XML. La posibilidad de añadir soporte para distintos mapas nos permite cambiar a voluntad los editores que se utilizan para crear estos mapas. Más adelante en la sección 6.7.2 explicamos cómo hemos creado los de para JV^{2}M.

Lo más importante del cargador de mapas es que éste debe ser independiente completamente del código específico. Una cosa es leer el mapa de disco y comprobar que su formato es correcto, y otra bien distinta es *lanzar* los objetos del juego que éste posee.

Una vez que el cargador ha comprobado que el fichero se adhiere correctamente al formato establecido, debe proceder a la creación de las entidades correspondientes. Una opción utilizada habitualmente es el uso de factorías extensibles, como la descrita en la sección 2.6.2. Sin embargo, el método más general es simplemente invocar a una función del código específico a la que

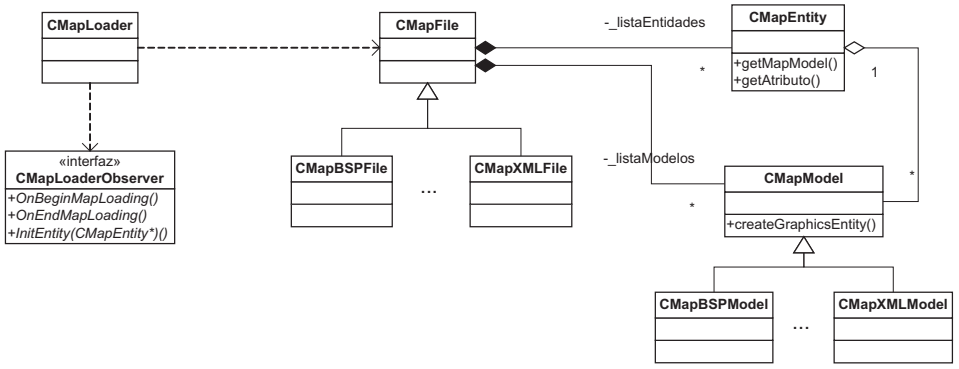


Figura 6.11: Diseño de clases del cargador de mapas

se le da toda la información de la entidad contenida en el mapa. El código específico entonces decidirá crear los objetos de juego correspondientes.

Esta independencia entre el cargador de los mapas y el código específico es una de las características que permitirán posteriormente configurar el entorno virtual en base al ejercicio utilizado, como se describe en el apartado 6.6.2.

El diagrama de clases utilizado en el cargador de mapas es mostrado en la figura 6.11. En él, vemos que existe una clase `CMapFile` que representa la información contenida en un mapa o nivel leído de disco. La clase contiene un método para construir un objeto de la clase a partir del nombre del fichero. Para soportar los distintos ficheros de mapa soportados, se utilizan clases derivadas creadas específicamente para cada uno de ellos. De esa forma forma el único punto de la aplicación que habría que modificar si se cambia el *pipeline* de contenidos (ver apartado 6.7) y se generan los mapas con otro formato, es, precisamente, el cargador.

Cada mapa leído de disco dispone de una lista de entidades, y una lista de modelos. Ya explicamos en la sección 5.8.2 la razón de que el propio mapa incluya información de modelos: algunas entidades tienen modelos estáticos (arquitectura del nivel) o representan el terreno. En vez de hacer referencia a modelos en ficheros externos distintos al del mapa, se empaqueta tanto la información de las entidades como de los propios modelos en un único fichero.

En el diseño de clases, la información de cada entidad se almacena en un objeto de la clase `CMapEntity`, que contiene (i) la lista de pares atributo-valor en la que se basará el código específico para crear los objetos de juego y (ii) el modelo asociado a esa entidad, si éste existe.

Por último, para representar el modelo, disponemos de la clase `CMapModel`, para representar el modelo guardado en el mapa asociado a una entidad. Cada tipo de mapa soportado tiene asociada una clase derivada de ésta, que es capaz de construir la entidad gráfica asociada a ese modelo a partir de la

información contenida en el fichero.

Para el soporte de ficheros `bsp`, hemos hecho uso de una librería creada por Valve[9]. El formato del fichero permite *empaquetar* tanto la lista de entidades como el modelo de los objetos estáticos del mundo ("arquitectura" y objetos como puertas, interruptores, papeleras o farolas). Por lo tanto, en este caso un objeto `CMapModel` representa un modelo cargado desde el propio `bsp`. El método que construye la entidad gráfica crea un objeto de la clase `Graphics::CBSPModelEntity` vista en la sección 6.3.1.

Por su parte, en el soporte para mapas contenidos en XML, esos modelos estáticos *no* se almacenan en el propio XML, sino en ficheros de datos externos. Las entidades con modelos estáticos hacen referencia a esos ficheros, y el cargador de mapas los lee automáticamente al analizar el XML y crear los `CMapEntity` asociados.

Para la comunicación entre el cargador de mapas del motor de la aplicación y el código específico, existe un interfaz, `CMapLoaderObserver`, que tiene tres métodos de notificación de eventos relacionados con la carga, dos para indicar que ésta va a comenzar o ha terminado, y un tercero, el más importante, invocado una vez por cada entidad del mapa. El código específico, entonces, creará los objetos de juego asociados a cada una de las entidades. El código fuente más relevante aparece en la sección C.1.

6.3.4. Motor de sonido

Para la implementación del motor de sonido, hemos utilizado OpenAL[10] (Lengyel, 2006), que proporciona una API de audio multiplataforma desarrollada por Creative Labs para el renderizado eficiente de audio posicional y multicanal en tres dimensiones. La librería ha sido utilizada con éxito en juegos de renombre como Doom 3 y Quake, algunos títulos de la saga UnrealTournament o Escape from Monkey Island.

Igual que en el caso del motor gráfico, sobre OpenAL se crea el concepto de "entidad sonora" que el código específico puede manejar. Ese objeto "entidad" permite reproducir un sonido determinado, así como posicionar la fuente para permitir sonido tridimensional.

6.3.5. Motor de colisiones

El motor de colisiones realiza los cálculos necesarios para detectar si dos objetos del entorno virtual han colisionado, para que éstos reaccionen de forma coherente a las reglas del propio entorno.

El motor de colisiones mantiene, pues, una representación propia de cada uno de los objetos (lo que se suele denominar *malla de colisión*), con la que

[9] El estudio que desarrolló Half-Life.
[10] Disponible en `http://www.openal.org/`

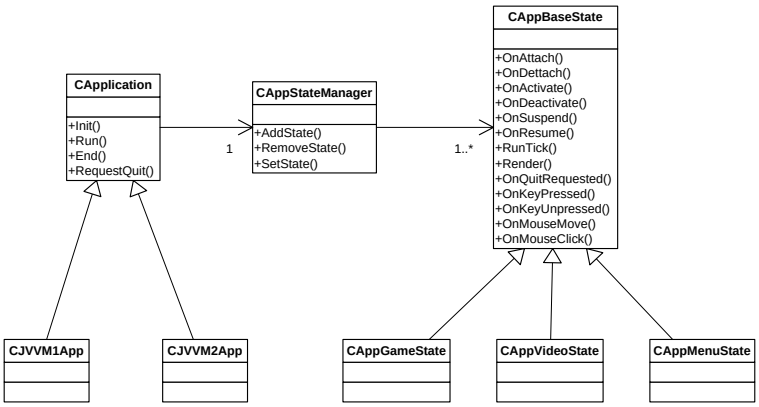


Figura 6.12: Diseño de clases de la aplicación

realiza sus cálculos, y que normalmente es una simplificación o aproximación *burda* del objeto que se dibuja.

Para el motor de colisiones hemos utilizado Opcode (*OPtimized COllision DEtection*), que también ha sido utilizado en otros motores de juego como CrystalSpace[11] o Irrlicht[12].

Igual que ocurre en todos los casos anteriores, lo importante para mantener la independencia entre éste y el resto de la aplicación, es proporcionar el concepto de *entidad de colisión* por encima de Opcode. El código específico puede crear este tipo de entidades para su beneficio. Por ejemplo, y en particular, la clase `CMapModel` descrita en la sección 6.3.3, al igual que dispone del método `createGraphicsEntity()` (ver figura 6.11), tiene un método que construye la entidad de colisión a partir del modelo contenido en el mapa.

6.3.6. Control de ejecución

El motor de la aplicación es el *dueño* de la hebra principal del programa, por lo que es el que dirige su ejecución completa. En particular, implementa el bucle principal del que se habló en las secciones 2.5 y 5.9.

Dado que la aplicación puede estar en *estados* distintos, el bucle principal ejecutado puede variar dependiendo del momento. Por citar un ejemplo, el bucle debe ejecutar unas tareas diferentes cuando está en el menú del usuario a las que invoca cuando está reproduciendo un vídeo a pantalla completa o en el propio juego.

Para facilitar esta gestión, la aplicación (`CApplication`) está implementada utilizando el patrón estado descrito por Gamma et al. (1995). El patrón convierte a la aplicación en una máquina de estados, en este caso gestionada

[11] Disponible en `http://www.crystalspace3d.org`.
[12] `http://irrlicht.sourceforge.net`

por la clase `CAppStateManager`. Los estados (`CAppBaseState`) *extraen* del bucle principal todas aquellas tareas específicas o dependientes de él. De esta forma, las tareas comunes a todos ellos (mantenimiento del gestor de memoria, procesado de eventos de ventana, etc.) pueden quedarse bajo el control de la aplicación, mientras que el resto son implementadas en clases externas que puede intercambiarse en tiempo de ejecución.

La idea puede incluso generalizarse aún más. Como se ve en la figura 6.12 en la implementación de JV^{2}M, existe una clase `CApplication` general de la que heredan las aplicaciones particulares. De esta forma, toda la gestión de los estados se puede reutilizar en distintas aplicaciones.

La diferencia entre las distintas aplicaciones (por ejemplo entre JV^{2}M 1 y JV^{2}M 2) está en:

- Métodos de inicialización y destrucción (`Init` y `End`): en ellos la aplicación inicializa todos los módulos o motores internos y cede el control al código específico para que haga lo propio. Si dos aplicaciones utilizan módulos distintos, su inicialización y destrucción variará[13]. También serán distintos los estados de la aplicación que construyen.
- Bucle principal (`Run`): en este método se invocan las distintas tareas generales, así como los métodos del estado activo. Si bien esta segunda parte es general y común a todas las aplicaciones, la primera puede necesitar modificaciones si las diferencias entre ellas son significativas.

En nuestro caso, tenemos dos clases distintas, una para cada versión de JV^{2}M. La única diferencia que existe entre ambas en la función de entrada al programa (`main`) es la clase a la que pertenece la única instancia creada de la aplicación:

```
int main(int argc, char *argv[]) {

    CJVVMApp theApp("JVVM");
    bool initCorrecto;

    theApp.SetCommandLineArgs(argc, argv);

    initCorrecto = theApp.Init();

    if (initCorrecto)
        theApp.Run();

    theApp.End();
}
```

[13] En ocasiones, no obstante, no hay diferencias a pesar de cambios en los motores, si éstos disponen por encima de una fachada como la explicada para el motor gráfico en la sección 6.3.1.

En cuanto a los métodos de los estados que son invocados por la aplicación distinguimos cuatro tipos:

- De inicialización y destrucción: cuando el estado es *enganchado* a la aplicación, es decir, cuando se añade como estado de la misma, y cuando es *desenganchado*, la aplicación invoca a los métodos correspondientes. En el de inicialización, el estado construirá las estructuras que necesitará durante la ejecución.
- De activación y desactivación: en general, los estados son creados al principio de la aplicación, de tal forma que la vida de estos objetos coincide con la ejecución del programa. No obstante, el estado no está *activo* durante todo ese tiempo, sino únicamente en ciertos momentos. Existen ciertas estructuras que no tiene sentido mantener en memoria cuando el estado está inactivo, por lo que la aplicación le *avisa* cuando éste es activado y desactivado. Esos métodos permiten al estado crear estructuras únicamente útiles durante el periodo de actividad, o realizar otras tareas concretas. Por ejemplo, el estado en el que se encuentra la aplicación cuando está mostrando el menú, puede lanzar la reproducción de sonido de fondo cuando se activa, y pararlo en su desactivación.
- De notificación de eventos: en particular, cuando la aplicación detecta eventos de entrada de usuario (pulsación de teclas, movimiento del ratón), éstos son redirigidos al estado particular para su gestión. Cuando se está en el estado del juego (`CAppGameState` en la figura), el propio estado se encarga de enviarlos a su vez a los gestores de la entrada descritos en la sección 6.3.2.
- De implementación de las tareas específicas del bucle principal: en JV^{2}M son dos, la ejecución de la lógica, y la de dibujado. Cada estado de la aplicación procederá en ellos de forma distinta. Por ejemplo, el estado del menú no implementa nada de lógica, y el dibujado consiste simplemente en mantener en pantalla las opciones, mientras que el estado del juego invoca al código específico en la primera, y dibuja las entidades registradas en el motor gráfico en la segunda.

De manera simplificada, el bucle principal final es el siguiente: en cada vuelta del bucle, la aplicación, por medio del gestor de estados, comprueba si debe realizar una transición; si es así, se llama a los métodos de desactivación y activación correspondientes. Posteriormente, actualiza el tiempo de la aplicación, para averiguar cuánto tiempo ha transcurrido. Después de realizar una serie de operaciones de mantenimiento del propio motor de la aplicación (gestor de memoria, procesado de los eventos de la ventana, etc.), comprueba la entrada de usuario e informa al estado activo si hay algún

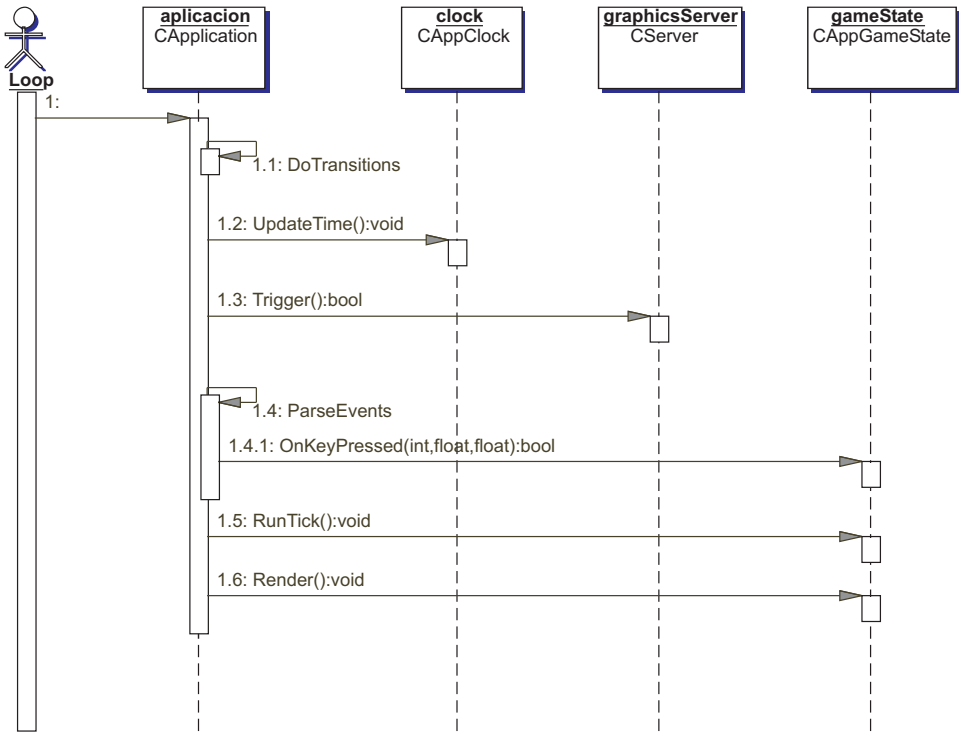


Figura 6.13: Diagrama de secuencia del bucle principal de JV^{2}M

evento. Acto seguido, invoca la actualización de la lógica implementada en el estado, y por último el método de dibujado. El proceso completo aparece en la figura 6.13[14].

Otra responsabilidad relacionada con el control de la ejecución enmarcado dentro del motor de la aplicación es la *gestión del tiempo*. Cuando el estado lanza la ejecución de su lógica debe conocer, como ya hemos dicho en otras ocasiones, cuánto tiempo ha transcurrido desde la última actualización, para que actualice sus entidades (posiciones, etc.), de acuerdo a ese tiempo[15].

Aunque en el diagrama del bucle principal ya aparecía la actualización del reloj, el diseño de clases involucrado merece una descripción detallada. Para minimizar el impacto ante cambios de plataforma y facilitar algunos aspectos que veremos enseguida, la gestión del tiempo está implementada de tal forma que *separa* el *reloj físico* o *fuente* de la que se lee la hora, y el reloj lógico que maneja la aplicación (ver figura 6.14).

[14] De todas las tareas de mantenimiento, únicamente aparece la de gestión del servidor gráfico.

[15] Esto es debido a que el bucle principal descrito es de tipo fotograma y paso de tiempo variable (descritos ambos en la sección 2.5).

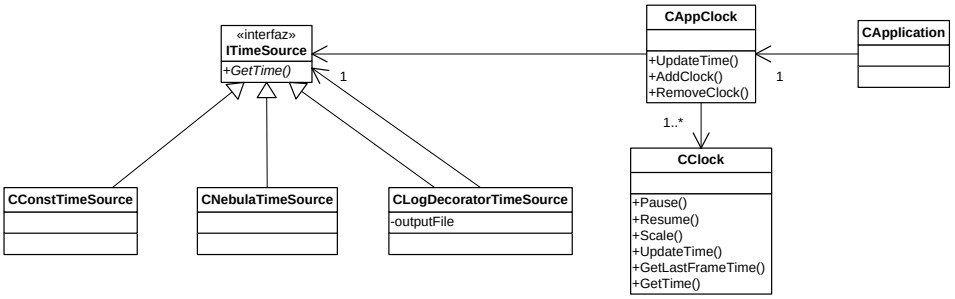


Figura 6.14: Diagrama de clases que gestionan la hora de la aplicación en JV^{2}M

La implementación, basada en Llopis (2004), tiene una clase abstracta que representa la *fuente* de la hora (`ITimeSource`). Dependiendo de la plataforma, la clase derivada usada utilizará las funciones de uno u otro sistema operativo. En nuestro caso, en vez de utilizar las funciones del sistema operativo subyacente, delegamos la responsabilidad a Nebula (la clase implementada es `CNebulaTimeSource`), que dispone de un reloj que también utiliza para actualizar los sistemas de partículas y otros de sus subsistemas.

La fuente de tiempo es la que dicta al reloj de la aplicación (`CAppClock`) la hora que debe regir la ejecución de cada vuelta del bucle. La actualización de la hora, que venía reflejada en la figura 6.13 en la invocación al método `CAppClock::UpdateTime`, delega en la fuente concreta para obtener el tiempo.

Esa hora almacenada en `CAppClock` sirve de *reloj base* para los distintos relojes *lógicos* que el motor de la aplicación permite. Esos relojes lógicos (`CClock`), permiten ser pausados e incluso escalados, de tal forma que avancen más rápido o más despacio que el reloj real. El código específico puede entonces basar su actualización en varios relojes lógicos a distintas velocidades, para conseguir, por ejemplo, el *tiempo bala*[16].

La separación entre la fuente de tiempo y el reloj de la aplicación permite, además, repetir ejecuciones de la aplicación para solucionar errores que parecen ocurrir de manera arbitraria, pero que en realidad están relacionados con las pequeñas variaciones de tiempo entre fotogramas que se producen en distintas ejecuciones de la aplicación. Utilizando el patrón decorador (Gamma et al., 1995), la propia fuente puede grabar a disco el tiempo en el que empieza cada fotograma (en la figura 6.14 aparecía en la clase `CLogDecoratorTimeSource`). En ejecuciones sucesivas, se puede utilizar una fuente de tiempo que simplemente lea de fichero la secuencia de horas de la ejecución anterior, eliminando el indeterminismo introducido por esas varia-

[16] También las entidades gráficas que aparecen en la figura 6.9 tienen referencias al reloj lógico que deben utilizar para actualizar sus animaciones.

ciones de tiempo.

Una vez fijadas las responsabilidades del control de la ejecución (bucle principal y gestión del tiempo) y la forma de implementarlos, veamos cómo son utilizados en JV^{2}M. La aplicación completa tiene tres estados distintos implementados:

- Menú: que permite la selección del perfil del estudiante y comienzo del juego.
- Vídeo: para reproducir secuencias al principio de la aplicación, etc. Puede ser configurado para que al pulsar una tecla (es decir, cuando el bucle principal invoca a su método `OnKeyPressed`), termine la reproducción y salte a otro estado distinto.
- Juego: en su activación provoca la carga del mapa concreto que tiene el escenario, y su configuración dependiendo del estado del subsistema de tutoría. Cuando se invoca desde el bucle principal la ejecución de su lógica, se traslada al código específico, que se encargará de actualizar las entidades, la vista lógica y el subsistema de tutoría. Para el dibujado, confía en el motor gráfico, donde se mandan dibujar todas las entidades gráficas registradas por el código específico. Cuando recibe eventos de entrada, los redirige a los gestores (`CKeyboardManager` y `CMouseManager`) que informan a las partes del código específico interesadas.

La parte más importante desde el punto de vista arquitectónico, sin duda, es qué mecanismos proporciona el motor de la aplicación (y en particular el control de la ejecución) al código específico para ser invocado en cada vuelta del bucle. En nuestro caso, disponemos de dos mecanismos:

- Invocación en cada vuelta del bucle (o fotograma): el código específico puede *engancharse* para ser invocado por éste en cada vuelta del bucle principal, justo antes del dibujado. La forma de hacerlo es mediante un observer. Este mecanismo es utilizado por módulos generales del código específico, como veremos en la sección 6.4.1.
- Temporizador: ya hemos mencionado alguna vez que los objetos del juego deben *actualizarse* cada cierto tiempo, para que tomen decisiones sobre su comportamiento. La opción más fácil es invocar un método de actualización en cada vuelta del bucle. Sin embargo, la mayoría de estas entidades no requieren tanta frecuencia para poder dar una sensación de credibilidad. Por eso, por eficiencia, el motor de la aplicación proporciona un temporizador, que permite a las entidades registrarse de tal forma que son avisadas cuando transcurren un número determinado de milisegundos. El diseño de clases que lo permite se muestra en la figura 6.15.

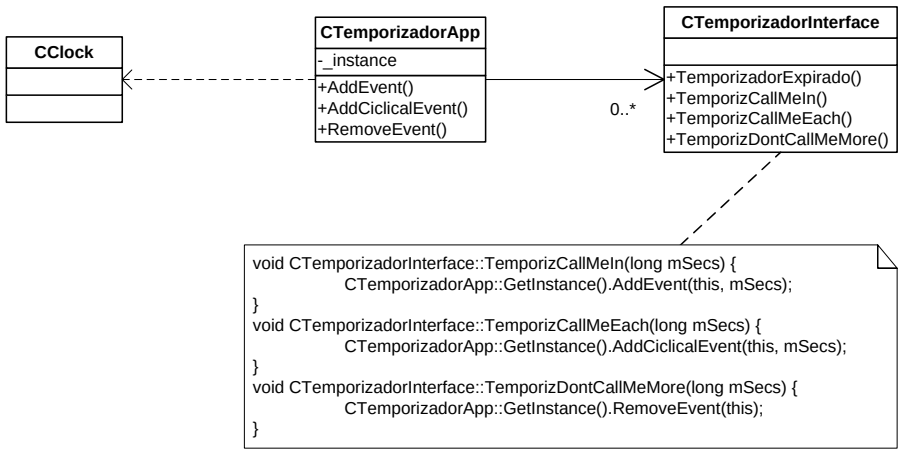


Figura 6.15: Diseño de clases del temporizador de la aplicación en JV^{2}M

6.4. Gestión de entidades

Como indicábamos en la sección 5.6, en este módulo se sitúan las entidades del juego, que representan los objetos *dinámicos* del entorno virtual (ya sean visibles o invisibles).

Algunas de estas entidades serán los representantes en el entorno virtual de los objetos que aparecen en la *vista lógica* que describiremos en la sección 6.5. Debido a esto, en la implementación de JV^{2}M hemos llamado a este sistema el *módulo gestor de los objetos del interfaz* o módulo OIM (*Object Interface Manager*).

6.4.1. Funciones principales

En la arquitectura para videojuegos educativos descritas en el apartado anterior, situábamos el control de las entidades como el *comienzo* de lo que en juegos vinimos a llamar *código específico*.

En general, se entiende que la principal función de esta capa es el control de todas las entidades que dan vida al juego. Sin embargo, desde el punto de vista de nuestra metodología y arquitectura, la función más importante es la *configuración* del entorno virtual con respecto al ejercicio.

Ya vimos en los capítulos anteriores que uno de los aspectos más importantes de nuestra arquitectura es que permite la separación de los dos tipos de datos necesarios para el desarrollo del videojuego educativo: los contenidos de instrucción y los entornos virtuales. De esta forma se consigue que el mismo entorno virtual (costoso de construir) pueda ser reutilizado por distintos ejercicios y viceversa.

Pues bien, la funcionalidad que permite la modificación o adaptación del entorno virtual en base al ejercicio seleccionado se sitúa en este módulo de gestión de entidades. En concreto, en la inicialización de los objetos de juego dependientes del contexto educativo, éstos comprueban el estado de la vista lógica que fija el ejercicio y se configuran en consecuencia. Con esta estructura, el diseñador del nivel o entorno virtual no determina por completo el comportamiento final del mapa, sino que deja algunos de sus aspectos abiertos.

En el caso del ejemplo que explicábamos en el apartado 5.11 de entrenamiento de bomberos, el diseñador del nivel coloca entidades en el entorno virtual que simbolizan puntos donde pueden aparecer víctimas. El experto del dominio, por su parte, crea ejercicios concretos con una cantidad de víctimas determinadas. Cuando el mapa se lanza, la inicialización de las entidades que definen las localizaciones de las víctimas pueden crear los avatares correspondientes, o pueden, al comprobar que el ejercicio no requiere ningún afectado más, no crear ningún otro objeto de juego.

En JV^{2}M, existen varias entidades que se configuran en su inicialización en base al ejercicio actual. Por ejemplo, las entidades que representan la pila de *frames* (que en JV^{2}M 1 se traduce en un edificio volante y en JV^{2}M 2 en distintas plantas de una nave espacial) en su inicialización comprueban el número de registros de activación suspendidos; por su parte, la pila de operandos (figuras 6.3c y 6.4b) crea tantas cajas como existan en el momento del lanzamiento del mapa. En la sección 6.4.3 veremos un ejemplo concreto de configuración de entidades a partir del ejercicio, en particular, cómo las manadas configuran su número de individuos dependiendo del método en ejecución.

Para la gestión de todas las entidades, tanto las que se configuran a partir del ejercicio actual como las que no, la capa OIM dispone de una lista de todos los mapas que hay lanzados en la aplicación. La necesidad de tener *más de un* mapa se debe a que, en el contexto educativo y la metáfora seleccionada, es posible que tengamos *varios* mapas activos a la vez. Por ejemplo, en JV^{2}M 2 existe un mapa para el nivel que representa un *frame*, y otro para la sala de una clase cargada; incluso se da el caso de tener varios mapas lanzados que provienen *del mismo fichero*, pero que han sido *configurados* de forma distinta. Por ejemplo, todos los niveles de la nave que representan *frames* o todas las salas que representan clases.

Dentro de la lista de mapas, existe uno que es el mapa actual en el que se encuentra el usuario. Pasar de un mapa a otro puede ocurrir o bien mediante *teletransportadores* (entidades especiales que lanzan el cambio de mapa con su uso), o bien al salir fuera de las fronteras físicas del mapa (por ejemplo, en JV^{2}M 1 hay un cambio de mapa cuando el usuario atraviesa la puerta que da acceso al *frame* actual que aparecía en la figura 6.2a).

Cada uno de los mapas tiene una lista de todas las entidades u objetos del

juego que contiene. La capa OIM crea el *mapa OIM* cuando el motor de la aplicación le informa (mediante el observer `CMapLoaderObserver` mostrado en la figura 6.11) del comienzo de la carga de un mapa desde disco[17]. A partir de ese momento, todas las entidades que se inicialicen se crearán en el mapa que se está lanzando, hasta el momento en el que el cargador de mapas del motor de la aplicación indique que ha terminado la carga.

Con esto, terminamos la descripción de las funciones más importantes de la capa OIM: gestión de las entidades que conforman el entorno virtual y creación y configuración de las mismas en base al mapa y al ejercicio actual. Sin embargo, para definir el comportamiento de las entidades, la capa OIM dispone de otros módulos, que pasamos a describir en la sección siguiente.

6.4.2. Módulos OIM

Además del control de los mapas y sus entidades, el módulo OIM contiene otra serie de submódulos de soporte que facilitan la programación de esas entidades. Estos submódulos no se sitúan dentro del motor de la aplicación porque su funcionamiento concreto es *específico* del juego en cuestión. Para su ejecución por tanto, deben ser invocados por éste, utilizando alguno de los dos mecanismos comentados previamente: activación de un temporizador, o en cada vuelta del bucle mediante un observer.

En el caso de JV^{2}M, existen tres módulos auxiliares de importancia:

- Percepción: en JV^{2}M 1 se encarga fundamentalmente de controlar las opciones que tiene disponibles el usuario cuando se acercaba a los distintos elementos activos del entorno (ver figura 6.3c). Por lo tanto, implementa la *percepción del estudiante*: cuando éste tiene dentro del campo de visión algún elemento activo, avisa a la entidad para que actúe en consecuencia (añadiendo la opción en la lista de opciones). En JV^{2}M 2 es útil para la programación de los NPCs.

- Búsqueda de rutas: para cada mapa lanzado, este módulo almacena un grafo con nodos. Las entidades pueden preguntar cómo ir de una posición a otra del entorno y el módulo, utilizando A^*, calcula el camino. Se utiliza, por ejemplo, en JV^{2}M 1 para que Javy, el agente pedagógico, encuentre el camino hacia el estudiante.

- Gestión de manadas: ya hemos nombrado las manadas como ejemplo de entidades que se configuran en base al ejercicio. Para determinar el modo en que se comportan sus individuos una vez creados, utilizamos las ideas expuestas por Reynolds (1999) usando un módulo externo, de

[17] Aunque la creación del mapa no se muestra en el diagrama de la secuencia de la figura 6.20, sí puede verse las invocaciones desde el cargador de mapas para avisar del comienzo y fin de la carga del mapa

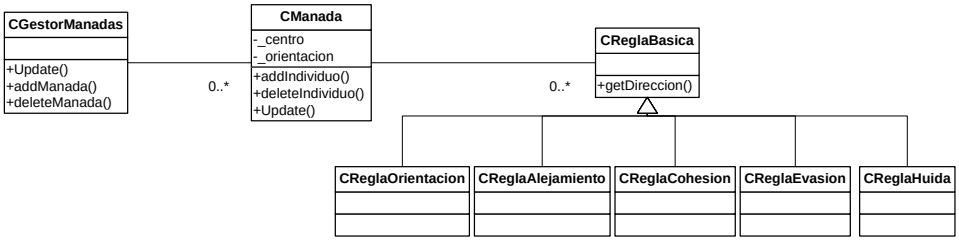


Figura 6.16: Diagrama de clases del gestor de manadas de JV^{2}M

tal forma que luego las entidades establecen su posición de acuerdo a lo que éste dicta.

Para que estos módulos funcionen *también* deben ser inicializados de acuerdo a la información contenida en el mapa durante la carga, de forma que algunas de las entidades *del mapa* no se convierten en objetos de juego, sino en información de inicialización de estos módulos.

Como ejemplo detallado de esto, veamos cómo funciona el gestor de las manadas, cuyo diagrama de clases aparece en la figura 6.16. Como se puede ver, el gestor (`CGestorManadas`) contiene una lista con todas las manadas (`CManada`) que gestiona. Por su parte, cada manada está formada por una lista de individuos, de los que guarda sus posiciones y orientaciones. Esas posiciones y orientaciones se deciden de acuerdo a unas *reglas* concretas, cada una con un peso específico diferente. En nuestro caso, existen cinco reglas distintas, que intentan, por ejemplo, que los individuos tiendan a acercarse al centro de la manada y a tener la misma dirección que ella, a no acercarse demasiado al resto de individuos o intentar evitar al jugador. Para poner todo esto en funcionamiento, el gestor de manadas es invocado en cada vuelta del bucle desde el motor de la aplicación, lo que provoca la actualización de todas las manadas y, por consiguiente, de todos sus individuos.

En JV^{2}M 2, además, las manadas siguen una ruta determinada. En particular, existe por cada manada un *grafo* por el que patrulla. Para ello, el centro de la manada se va moviendo por las distintas aristas del grafo, haciendo que todos los individuos de la manada se muevan en esa dirección. Para almacenar los grafos se utiliza un módulo auxiliar que utiliza los tipos de datos proporcionados por la librería `boost`[18].

Toda esta infraestructura para la gestión de las manadas, sin embargo, es inútil si no hay alguien que construya alguna manada cuando se carga un mapa. Para ello, el diseñador del nivel debe añadir al mapa dos tipos de información distintos:

[18] `http://www.boost.org/`

Figura 6.17: Manadas en funcionamiento en JV^{2}M 2

- El grafo por el que patrullan las manadas: consiste en colocar entidades puntuales que marcan las posiciones de los vértices (entidad `waypoint`), y una entidad que define las aristas que tiene el grafo (entidad `waypointGraph`). El grafo viene dado mediante una cadena con las aristas codificadas utilizando los nombres de los vértices[19].

- La información de la manada: mediante otra entidad distinta (`manada`). Ésta contiene algunos de los atributos generales de la manada, como los modelos gráficos que utilizan sus individuos, o el nombre del grafo por el que patrullan.

Cuando el cargador de mapas del motor de la aplicación lee esas entidades, invoca al método `CreateEntity` del módulo OIM (gracias al interfaz de la figura 6.11 explicado en la página 171). Para las entidades `waypoint` y `waypointGraph`, se registra la información asociada del grafo en el módulo correspondiente, que será procesada al finalizar la carga del mapa (`OnEndMapLoading`), ya que hasta ese momento no se garantiza que todos los vértices se hayan leído. Por su parte, la entidad `manada` crea y registra la manada en el gestor, crea la "inteligencia artificial" de la manada, que moverá su centro por el grafo y crea sus individuos. Estos dos últimos pasos son descritos con detalle en la sección 6.4.7. El resultado final puede verse en la figura 6.17, donde aparece una captura de la versión de depuración de JV^{2}M 2 con líneas azules representando el grafo que sigue la manada, y

[19] La codificación concreta es una lista de parejas, cada una compuesta por un vértice, y una lista con sus vértices adyacentes, de forma que, al ser un grafo bidireccional o no dirigido, se pueden omitir componentes. Por ejemplo, la cadena "`A{B C}B{C}`" representa el grafo de tres vértices, todos ellos interconectados.

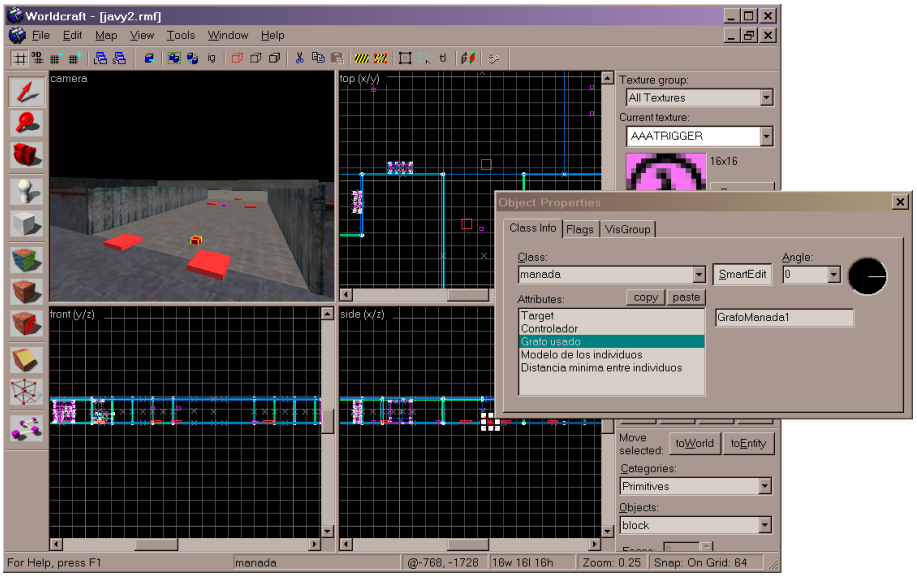


Figura 6.18: Edición de manadas en Worldcraft

líneas blancas que parten del centro de cada uno de los individuos hasta el centro de la manada.

6.4.3. Configuración de entidades según el ejercicio

Una vez detallado el módulo de gestión de las manadas, podemos ejemplificar el proceso que siguen para configurar sus propiedades en base al ejercicio.

Como ya comentamos, en JV^{2}M 2 el jugador tiene que conseguir los recursos necesarios para ejecutar el método. Cada recurso está oculto en un miembro de la manada, por lo que el número de individuos que ésta posee inicialmente depende del ejercicio.

Ya hemos visto que para que las manadas aparezcan en el nivel, el diseñador ha tenido que colocar en el mapa los puntos por donde ésta paseará, así como una entidad `manada` que es la responsable, precisamente, de la creación de los individuos. En la figura 6.18 aparece una captura en el momento de la edición; en ella, se puede ver seleccionada la entidad responsable de la creación de las manadas (cubo seleccionado en la vista de la esquina superior izquierda), junto con sus propiedades. El editor después graba un fichero que el motor de la aplicación, y en particular el cargador de mapas, leerá durante la ejecución; la figura 6.19 muestra la parte relacionada con las manadas.

Ya en tiempo de ejecución de la aplicación, el proceso de configuración de las manadas, que termina con el número de individuos necesarios para el

```
...
<entity classname="waypoint" targetname="A1"
        origin="-84 -1456 -4"/>
<entity classname="waypointGraph" origin="-111 -1775 0"
        graph="B1{A1 E1 D1}C1{A1 D1 E1}F1{D1 G1}G1{E1}"
        targetname="GrafoManada1"/>
<entity classname="manada" control="waypoints" radio="128"
        grafo="GrafoManada1" model="alien/alien.n2"/>
...
```

Figura 6.19: Código parcial de un mapa

ejercicio (figura 6.17), es el siguiente:

- El subsistema de tutoría selecciona el siguiente ejercicio a presentar al usuario, utilizando el perfil del estudiante y la base de ejercicios. En nuestro caso, el ejercicio está compuesto, entre otras cosas, del código en Java de una o más clases.
- La vista lógica del mundo se configura en base a ese ejercicio. En JV^{2}M, se utiliza la descripción del ejercicio para configurar la máquina virtual: se cargan las clases con sus métodos, y se lanza la ejecución del programa hasta el punto en el que el estudiante debe comenzar a ejecutar.
- El mapa con el entorno virtual es cargado. Aquí entra en juego el motor de la aplicación, en particular, el cargador de mapas descrito en la sección 6.3.3, que va leyendo todas las entidades.
- Una de esas entidades será la entidad `manada` que aparece en la figura 6.19.
- La lectura de esa entidad provoca la creación de un nuevo objeto de juego responsable del control de la manada (se verá en detalle este control en el apartado 6.4.7).
- En la inicialización de la entidad, el objeto pregunta a la vista lógica (máquina virtual) qué recursos son necesarios para la ejecución del método actual (por ejemplo, si el método tiene una instrucción del tipo "`c = 3 + 5;`", los recursos serán tanto las constantes 3 y 5 como el nombre de la variable `c`.
- El control de la manada crea un individuo por cada recurso.

El proceso completo puede verse a modo de diagrama de secuencia en la figura 6.20.

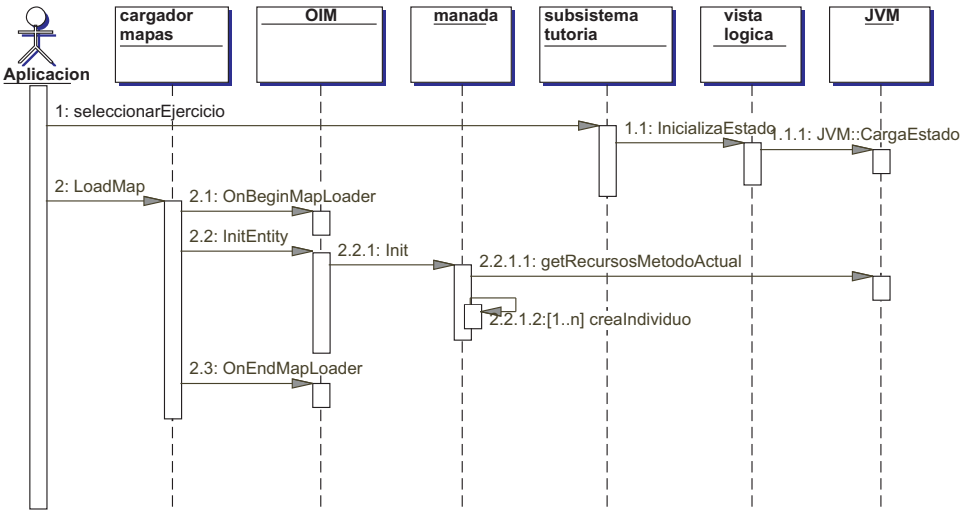


Figura 6.20: Diagrama de secuencia de la inicialización de una manada

6.4.4. Objetos OIM

Las entidades del juego, o como los hemos llamado en ocasiones en los capítulos anteriores objetos del juego (*game objects*), están implementados en JV^{2}M en la clase `OIMObject`. Objetos de esta clase existen para representar los NPCs que se mueven por el entorno, así como el resto de componentes activos (visibles o no), como trampillas o terminales.

Para la gestión de estos objetos, la capa OIM dispone de una clase que almacena toda la información relativa a un *mapa lanzado* (`OIM::CMap`). En particular, contiene:

- El reloj de la aplicación (`CClock`, ver sección 6.3.6) que se utiliza como base para todas las entidades de ese mapa.
- Los mapas gráfico y de colisión, es decir las referencias a objetos del motor de la aplicación, donde se añaden las entidades gráficas y de colisión de todos los `OIMObject` de ese mapa.
- El grafo utilizado para la búsqueda de caminos de los NPCs en ese mapa. El grafo se inicializa de forma análoga a los grafos de las manadas explicados en la sección anterior.
- Los grafos de las manadas, así como el gestor de manadas utilizado en ese mapa.
- Dos listas de objetos OIM, una que contiene las entidades "activas" y otra con las "inactivas" o suspendidas. La diferencia fundamental entre

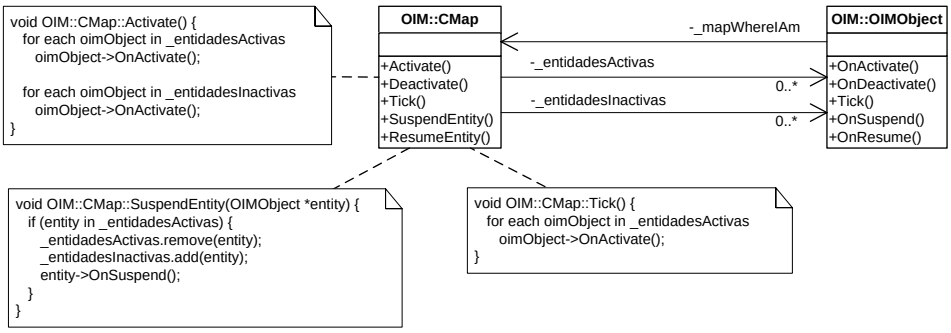


Figura 6.21: Relación entre mapas y objetos OIM

ambas es que las entidades suspendidas no necesitan ser actualizadas en cada vuelta del bucle, sino que están esperando a que se produzca algún evento, como la expiración de un temporizador.

Todas estas estructuras son inicializadas cuando el motor de la aplicación indica que se va a proceder a la carga de un nuevo mapa. Si alguna de ellas requiere cálculos posteriores (como por ejemplo precálculos relacionados con los grafos de las manadas del mapa), éstos son realizados cuando el motor de la aplicación informa de la finalización.

Una vez lanzado el mapa, la interacción con el entorno virtual puede comenzar. En ese momento, el bucle principal de la aplicación estará realizando todas las tareas de mantenimiento y en cada vuelta del bucle invocará a un método `Tick` de la capa OIM y a todos los objetos cuyo cronómetro haya expirado.

En cada `Tick`, la capa OIM invoca a los módulos internos (en particular, gestión de manadas y percepción). Acto seguido, cede el control al objeto que representa el mapa actual, el cual recorre toda la lista de entidades activas (`OIMObject`), llamando a sus métodos de actualización respectivos; el esquema del proceso puede verse en la figura 6.21.

Relacionado también con la gestión de los mapas, ya hemos dicho que puede haber varios lanzados pero solamente uno activo cuyas entidades se actualizan. Cuando se produce un cambio de mapa activo, el módulo OIM invoca a los métodos `Activate` y `Deactivate` del mapa activado y desactivado respectivamente, que a su vez avisan a todas sus entidades para permitirles realizar acciones concretas, como registrarse o deregistrarse del cronómetro de la aplicación.

Aparte de servir de *contenedor* de entidades, los mapas de la capa OIM también son utilizados por las propias entidades para buscarse entre sí y poder comunicarse entre ellas. Esta comunicación es indispensable para con-

seguir la sensación de que el entorno reacciona de manera coherente a sus propias reglas. Por ejemplo, un interruptor puede activar un ascensor, o el propio ascensor, al llegar a su destino, puede abrir las puertas situadas en el piso alcanzado.

Para poder hacer esto, existe un mecanismo de comunicación entre entidades, utilizando *señales*. En juegos con interacción sencilla y clara, como ocurre en JV^{2}M 1, esas señales quedaban reducidas a cinco, por lo que para la implementación del sistema de envío se ha optado por la creación en la clase `OIMObject` de los cinco métodos correspondientes. La lista concreta de las cinco señales posibles enviadas entre entidades aparece a continuación; de ellas, las tres últimas sólo pueden ser generadas por la entidad del jugador o la entidad de JAVY, el agente pedagógico[20]:

- `Touch`: lanzada cuando una entidad *toca* a otra. Habitualmente es la entidad del estudiante/jugador la que provoca esta señal, aunque no es necesario. Se utiliza en aquellas entidades donde el contacto provoca la ejecución de una acción. Por ejemplo, la forma de detectar que el jugador sobrepasa la frontera de un mapa es colocando una entidad no visible en ella; cuando ésta detecta el contacto con el estudiante, ejecuta el cambio de mapa.

- `Use`: la idea inicial es lanzarla cuando el estudiante *utiliza* esa entidad, por ejemplo, al activar un interruptor. Sin embargo, su semántica se puede extender y utilizar para diversos fines. Por ejemplo, se puede utilizar para *hablar* a un personaje.

  La señal puede ser lanzada no solo desde el usuario, sino también desde otras entidades. Por ejemplo, un interruptor puede lanzar la señal a la puerta, para que esta se abra; a su vez, la puerta puede también *usar* una entidad emisora de sonido, para que reproduzca el efecto de las bisagras girando. Un último ejemplo es la entidad que representa a JAVY en el entorno; ésta genera la señal para interactuar con los objetos, de la misma forma que lo haría la entidad del jugador.

- `UseWith`: este tipo de señal sí es específico de la mecánica de JV^{2}M 1 y otras aventuras gráficas. La señal es enviada por el jugador cuando desea utilizar una entidad del entorno con otra que tiene en el inventario. Así, en este tipo de juegos, el usuario puede utilizar una llave con una puerta, una esponja con una bisagra[21] o una caja con la pila de operandos (ver figura 6.3c).

[20] Algunas de estas señales las detallábamos en la sección 5.7; en aquel momento las llamábamos *perceptores*, ya que conceptualmente se puede entender que una entidad *percibe* que otra entidad la toca, etc. En este apartado esa percepción la convertimos en una *señal* entre entidades.

[21] Como en Escape from Monkey Island.

- `Take`: lanzada por la entidad del estudiante cuando quiere *coger* el objeto para guardarlo en el inventario. No son muchas las entidades que responden a esta señal; únicamente lo hacen aquellos objetos interactivos que permiten almacenarse en el inventario, como la caja situada encima de la pila de operandos.

- `LookAt`: igual que en los dos casos anteriores, este tipo de acción es específica de las aventuras gráficas. En el caso de JV^{2}M 1, la señal, generada por la entidad del estudiante, se utiliza cuando éste realiza la acción de *mirar* al objeto del entorno. La acción que la entidad realiza suele ser escribir una pequeña explicación en la pantalla relativa al objeto al que representa. Por ejemplo, la figura 6.3b muestra la explicación que aparece cuando el estudiante *mira* a Framauro, el personaje que se utiliza para crear y destruir *frames*.

En juegos con mecánicas más complejas, no obstante, las señales de comunicación son un mecanismo más general que no puede reducirse a cinco métodos. En esos casos, la información de una señal es almacenada en una estructura, construida por la entidad que lanza la señal. Para el envío, se utiliza un método genérico que recibe como parámetro la información de esa señal, donde se analiza y se actúa en consecuencia. La implementación de este envío y recepción de señales es similar al que describimos en la sección siguiente, cuando hablamos de los componentes.

6.4.5. Implementación mediante componentes

Ya indicamos en el capítulo anterior que este módulo distingue entre tres tipos de entidades: las entidades genéricas que no dependen del dominio, aquellas entidades relevantes para el contexto educativo, y la entidad que monitoriza el estudiante. En JV^{2}M, existen objetos de juego de los tres tipos:

- Entidades genéricas: por ejemplo, en JV^{2}M 1 existen entidades que representan localizaciones para las distintas cámaras que pueden "*pincharse*" durante el juego; en ambas versiones hay entidades para indicar dónde empieza el jugador o el tránsito de qué zonas provocan el cambio de mapas; por último, en JV^{2}M 2 hay entidades que permiten al jugador teletransportarse a otras localizaciones, y puertas que dan acceso a ascensores.

- Entidades específicas del dominio: en ambas versiones hay objetos que representan a la pila de operandos, valores concretos, objetos o clases. La mayoría de estas entidades son *observadoras* de la vista lógica de la JVM implementada en el módulo descrito en la sección 6.5, de tal forma que los cambios que suceden en ella se reflejan en el entorno virtual.

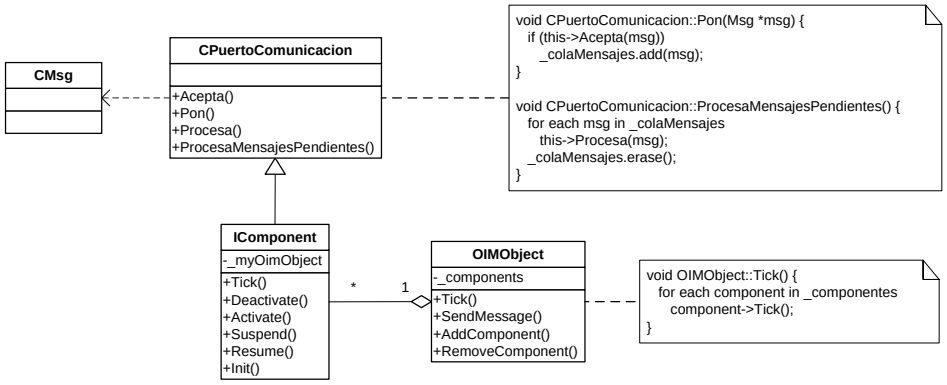


Figura 6.22: Diseño de clases involucradas en la implementación mediante componentes

- Entidad de monitorización del estudiante: la entidad responsable de la representación del avatar en el entorno virtual es también responsable de esta monitorización, avisando a la vista lógica y en última instancia al subsistema de tutoría sobre eventos significativos, como el comienzo de una conversación con JAVY.

La implementación de todas estas entidades se realiza utilizando componentes. Cada entidad (`OIMObject`) está formada por una lista de componentes (`IComponent`) capaces de recibir mensajes y reaccionar ante ellos. Cuando el motor de la aplicación pide al módulo la inicialización de una nueva entidad (durante la carga de un mapa mediante el método `InitEntity` visto en la figura de la página 170), el *lanzador* de las entidades comprueba, a partir de la clase de la entidad, qué componentes es necesario crear[22]. La inicialización de esos componentes se realiza de acuerdo con los datos cargados desde el mapa y guardados en la clase `CMapEntity` que aparecía en la misma figura.

Las clases principales involucradas en la implementación por componentes aparece en la figura 6.22. Aunque la parte más importante es la clase `IComponent` empezaremos nuestra descripción con la forma de enviar mensajes ya que, como hemos indicado anteriormente, el funcionamiento de los componentes se basa en ellos para la comunicación.

En el diseño de clases, existe una clase genérica, `CMsg`, de la que heredan todos los tipos de mensaje. Cada una de las clases nuevas añade los atributos o datos asociados al mensaje; por ejemplo, la clase `CUpdatePosition` que representa un mensaje enviado a los componentes para que actualicen la posición de la entidad tiene como atributo la información de la nueva posición.

[22]Para averiguarlo, se utiliza un fichero XML externo.

Para descubrir, dado un mensaje, a qué clase pertenece, existe un mecanismo parecido a RTTI que permite averiguar si un determinado mensaje es de una clase o no.

Como mecanismo de comunicación, cualquier clase que quiera ser capaz de recibir mensajes, debe heredar de la clase `CPuertoComunicacion`. Su método `Pon` implementa el *envío diferido de mensajes*, añadiendo el mensaje que recibe como parámetro a una cola con todos los pendientes por procesar, que son analizados cuando se invoca al método `ProcesaMensajesPendientes`. Las clases derivadas deben sobreescribir el método `Acepta`, que indica si la clase está interesada en recibir un tipo de mensaje concreto, y `Procesa` que recibe un mensaje como parámetro y actúa en consecuencia[23].

Los componentes, para poder ser receptores de los mensajes, heredan precisamente de esta clase[24], de tal forma que cada uno implementa los métodos `Acepta` y `Procesa` de forma distinta.

La clase `IComponent` contiene además otros métodos importantes para la gestión de la entidad. En particular, todos los métodos que aparecían en el `OIMObject` de la sección anterior (como `Activate` o `Tick`), están también implementados en cada uno de los componentes. Cuando el objeto del juego es invocado lo redirige a todos sus componentes, para que actúen consecuentemente.

La clase `OIMObject` es además extendida con otros métodos de gestión de los componentes, como los habituales para añadir y eliminar componentes, así como métodos que permiten enviar mensajes a todos ellos, y que son utilizados por los propios componentes cuando quieren indicar algún evento a sus hermanos.

En la siguiente sección mostramos algunos ejemplos de componentes utilizados en JV^{2}M, con sus mensajes respectivos. Posteriormente, mostramos algunos ejemplos de uso de estos componentes para conseguir el comportamiento de algunas entidades concretas.

6.4.6. Ejemplos de componentes

En JV^{2}M existen numerosos componentes, todos ellos heredando de la clase `IComponent`, que son agrupados durante la ejecución en los distintos `OIMObject` para crear el comportamiento completo de cada entidad del juego.

De forma general, podemos distinguir los siguientes tipos de componentes, dependiendo de su función:

- Componentes que dan acceso a distintos motores o proporcionan atri-

[23] Dado que `Procesa` es un método público, también sirve para enviar a un objeto un mensaje que será procesado *inmediatamente*, en vez de esperar al análisis de todos los mensajes.

[24] Esto provoca que, a pesar de lo que parece indicar el nombre `IComponent`, la clase *no* es un interfaz, sino una clase con algunos métodos implementados.

butos a la entidad: añaden a la entidad a la que representan por ejemplo la posibilidad de dibujar un modelo o generar un sonido. Estos componentes manejan entidades de alguno de los motores de la aplicación (gráfico, sonido, colisión) y alteran sus propiedades dependiendo de los mensajes recibidos. También situamos aquí los componentes que añaden atributos concretos al objeto del juego. Estos componentes tienen una importancia vital, ya que en la sección anterior vimos que los `OIMObject` son meros contenedores de componentes. Por lo tanto, si queremos por ejemplo que nuestra entidad tenga una posición o un nivel de salud, debemos añadirle sendos componentes que se responsabilicen de ellos.

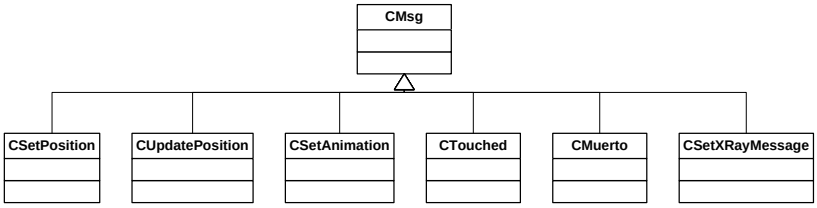
- Componentes que dan acceso a la vista lógica: ya salía alguno en la figura 5.4 de la página 136. Son aquellos que o bien reciben eventos de la vista lógica para actualizar la representación visual, o bien la invocan para cambiar alguna de sus propiedades.

- Gestión de la entrada: añade a un objeto la habilidad de procesar la entrada del usuario y reaccionar ante ella. El componente, además de heredar de `IComponent` suele registrarse en el motor de la aplicación para recibir los eventos de la entrada, según lo explicado en la sección 6.3.2. Normalmente este tipo de componentes se asigna a la entidad que controla el avatar del jugador. Cuando detecta pulsaciones de tecla o movimientos del ratón, manda los mensajes de cambio de posición correspondientes al resto de componentes.

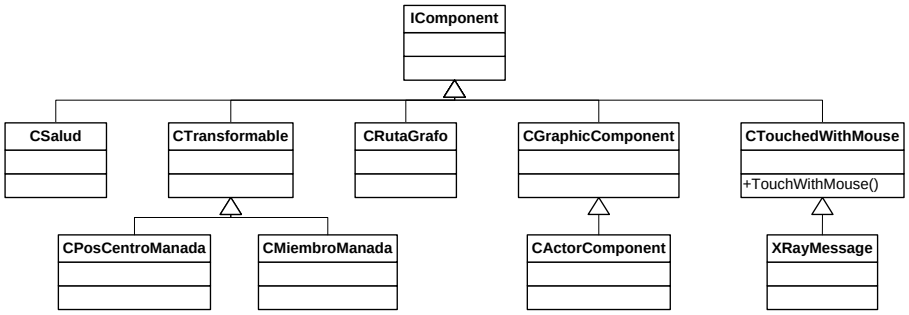
- Componente de inteligencia artificial específicos: los distintos NPCs tienen distintas estrategias de comportamiento. Por ejemplo, el avatar que representa a Javy delega su comportamiento de alto nivel a la vista lógica, por lo que el componente sirve de comunicación entre ambos módulos. El comportamiento de otros NPCs está implementado con máquinas de estado, utilizando un componente específico. Cuando se decide qué acción realizar, el componente de comportamiento envía mensajes al resto.

- Componente de seguimiento de rutas: utilizado por los NPCs para ir de un sitio a otro. Cuando la inteligencia artificial decide moverse a un punto en el mapa, envía un mensaje que es recogido por este componente, que durante una serie de fotogramas va moviendo a la entidad por el camino calculado[25]. Uno de estos componentes es el responsable de la implementación de la *función latente* `moveTo` que aparecía en la sección 2.7.3.3 cuando hablábamos de lenguajes de script.

[25] Enviando los mensajes de cambio de posición correspondientes.



(a) Jerarquía parcial de los mensajes entre componentes



(b) Jerarquía parcial de componentes en JV^{2}M 2

Figura 6.23: Jerarquía de clases de componentes y mensajes en JV^{2}M 2

Del primer tipo de componentes el más significativo es `CTransformable`, que añade al `OIMObject` asociado un atributo que indica su posición y orientación. Ambos datos son convenientemente inicializados con la información leída del mapa. Durante la ejecución, si algún componente (por ejemplo el encargado de la IA) desea cambiar la posición de la entidad, debe enviar un mensaje del tipo `CSetPosition` para que éste componente la reciba. Acto seguido, él informará al resto de componentes del cambio efectivo de la posición, con `CUpdatePosition`.

Uno de los componentes que reaccionan ante este mensaje es el responsable del modelo gráfico de la entidad. Este componente, que ya aparecía en la figura 5.4 de la página 136 bajo el nombre "EntidadGrafica", ha sido implementado por la clase `CGraphicComponent`. En su inicialización, utiliza las propiedades del mapa para crear la entidad en el motor gráfico de la aplicación, utilizando la clase `Graphics::CEntity` (ver figura 6.9). Además del mensaje para cambio de la posición en el motor, el componente acepta un segundo tipo de mensaje para hacer visible o invisible la entidad gráfica, y un tercero para cambiar el modelo. El componente, además, es especializado por otras clases que permiten controlar entidades más complejas. Por ejemplo, `CActorComponent` añade la posibilidad de controlar entidades animadas (como actores) que permiten adjuntar otros modelos (para coger armas). Para ello, el método de inicialización crea una entidad gráfica animada (`Graphics::CAnimatedEntity`), y es capaz de procesar mensajes de para el cambio de animación y para añadir modelos a sus huesos.

En principio, ninguno de los componentes anteriores realiza ninguna acción especial en cada vuelta del bucle, a excepción de comprobar su cola de mensajes para operar. Como un ejemplo de un componente que realiza alguna operación más es `CMiembroManada`, un componente que aparece en las entidades que representan personajes de una manada. En su inicialización, recibe la manada a la que pertenece (`CManada` en la figura 6.16) y el número de individuo al que representa. En cada actualización, pregunta al gestor de la manada en qué posición debe aparecer el personaje y la actualiza, enviando un mensaje del tipo `CUpdatePosition` a todos los componentes de la entidad.

Por último debemos poner un ejemplo de componente que, igual que `CGraphicComponent`, no hace nada en su actualización (`Tick`), pero es invocado por otros módulos distintos, mediante el interfaz correspondiente. Uno de estos componentes es `CVarGroupComponent`. En este caso, en la actualización no se realiza ninguna operación especial y ni siquiera se procesan sus mensajes pendientes, ya que no reacciona ante ninguno de ellos. La única utilidad de esta clase es hacer de *observador* de la vista lógica, de tal forma que cuando una variable local concreta de la máquina virtual cambia de valor, el componente es informado mediante la invocación de un método[26]. Cuando recibe la notificación, envía un mensaje que provoca el cambio del comportamiento de la caja en el entorno virtual, como veremos en la siguiente sección.

La figura 6.23 muestra la jerarquía parcial de los mensajes y componentes de JV^{2}M. Algunos de ellos han sido explicados en esta sección, mientras que otros aparecerán en la sección siguiente, cuando mostremos cómo funcionan tres entidades concretas de JV^{2}M 2.

6.4.7. Ejemplos de entidades de juego

Una vez que hemos ejemplificado algunos de los componentes implementados en JV^{2}M, describiremos la forma en la que se configuran tres de los muchos objetos de juego que aparecen en JV^{2}M 2. En nuestra descripción utilizaremos algunos de los componentes ya detallados, así como algún otro componente nuevo.

6.4.7.1. Caja que representa una variable local

En JV^{2}M los valores de la JVM se representan mediante cajas en el mundo virtual; esto es cierto tanto para los valores almacenados en la pila de operandos como para variables locales y atributos.

Centrándonos en las variables locales en JV^{2}M 2, éstas aparecen agrupa-

[26] Hablaremos más sobre la implementación concreta de la máquina virtual en JV^{2}M en la sección 6.5.

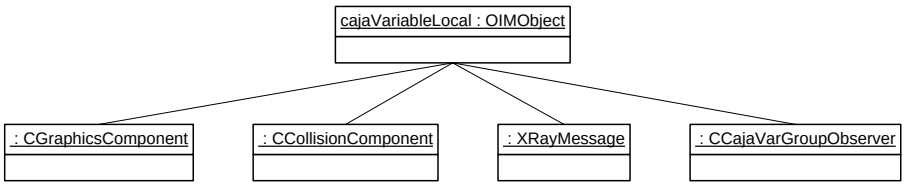


Figura 6.24: Componentes utilizados en las variables locales

das formando una pila de cajas, cada una de ellas simbolizando una variable local. El responsable de la pila de cajas es un objeto de juego (implementado también con su correspondiente colección de componentes) que recibe notificaciones de la JVM cuando se crean nuevas variables locales en el *frame* actual. Cuando esto sucede, el propio objeto del juego *crea* otra entidad nueva que representa la caja.

Desde el punto de vista del usuario, cada una de ellas es un simple elemento estático que no puede atravesarse y con una única particularidad: cuando se activa la visión de rayos X y el visor se sitúa encima, aparece un texto que indica su contenido.

Para conseguir este comportamiento, el objeto OIM que representa esta entidad dispone de los siguientes componentes (figura 6.24):

- Componente gráfico: en este caso, el modelo es una caja en la que aparece el tipo de datos que oculta (I para enteros, F para reales, etc.).

- Componente de colisión: que añade la entidad de colisión al motor correspondiente, de tal forma que el usuario no pueda atravesar la caja. Además, este componente es avisado por el motor de la aplicación cuando el puntero del ratón colisiona con ella. En ese momento, envía un mensaje del tipo `CTouched` al resto de los componentes de la entidad.

- Componente para rayos X: el mensaje anterior, `CTouched`, es procesado por un componente llamado `CTouchedWithMouse`. Cuando lo recibe, comprueba que el contacto ha sido con el ratón (utilizando las propiedades del mensaje), y en ese momento, invoca a un método de la propia clase que realiza una acción determinada. El método, inicialmente vacío, es sobreescrito por el componente `XRayMessage`, cuya misión es poner un mensaje de texto en el visor de Rayos X cuando se activa. El texto concreto que se escribe es configurable desde el exterior, utilizando un mensaje de tipo `CSetXRayMessage`.

- Componente observador de las variables locales: como explicaremos en la sección 6.5, la máquina virtual permite *observadores* que son notificados ante cambios de estado de la JVM. En concreto, permite

el registro de clases para ser avisadas ante cambios en los valores de las variables locales de un *frame*. Para que la entidad que representa una caja sea capaz de cambiar su comportamiento ante estos cambios, se configura con el componente `CCajaVarGroupObserver` que cuando recibe notificación de algún cambio hace dos cosas: envía un mensaje de cambio del mensaje de rayos X, para que el texto que aparezca refleje el nuevo contenido, y, en algunas ocasiones, envía un mensaje de *cambio de modelo*, cuando la variable ha cambiado el *tipo*[27].

Lo interesante de este ejemplo no es únicamente reflejar cómo se utilizan los distintos componentes para forma la entidad, sino ver la posibilidad de variación. Como ya hemos dicho, las cajas son ampliamente utilizadas en JV^{2}M. En particular, también se muestran cajas para reflejar el estado de la pila de operandos del *frame*, como puede verse en la figura 6.4b. Cada una de estas cajas tiene el mismo comportamiento que las de las variables locales, con la única diferencia de que su estado *no* cambia durante todo su ciclo de vida. Por lo tanto, su configuración de componentes es exactamente la misma que la citada anteriormente, pero sin el componente observador. Por su parte, las cajas que representan atributos de instancia o de clase tienen una configuración similar, pero el componente observador recibe notificaciones de cambios de valores en el atributo, en vez de en la variable local.

6.4.7.2. Inteligencia artificial de las manadas

En la sección 6.4.2 veíamos que para que aparezca una manada en el entorno virtual, el diseñador tenía que añadir al mapa varias entidades de distintos tipos (`waypoint`, `waypointGraph` y `manada`, todas ellas pueden verse en el código de la figura 6.19). También veíamos que para su control, se hace uso de las clases `CGestorManadas` y `CManada`, ambas dentro del gestor de manadas.

En esta sección describimos otro punto importante de las manadas, que ya salía en la figura 6.20: el objeto del juego que se crea cuando se carga el mapa. En particular, veremos cómo se realiza su implementación utilizando componentes.

La función de este objeto de juego es doble:

- Será el responsable de la inteligencia artificial de la manada: entendemos por inteligencia artificial la que establece las reglas de comportamiento que rigen a sus individuos (ver figura 6.16) y la que se encarga

[27] El cambio de modelo es necesario para que se cambie la apariencia de la caja y muestre el tipo correctamente.

Por su parte, una variable local en la JVM puede cambiar de tipo cuando el flujo de ejecución sale fuera del ámbito de esa variable, y entra dentro del ámbito de otra; la máquina virtual aprovecha el espacio físico de la primera para la segunda.

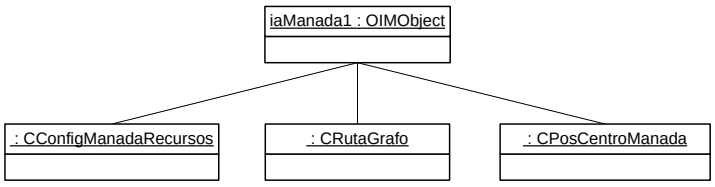


Figura 6.25: Componentes utilizados en la IA de las manadas

de *mover la manada completa*, haciendo que su centro siga la ruta del grafo con el que se ha configurado.

- Es el responsable de la creación de los individuos iniciales de la manada. Como dijimos en la sección 6.4.3, en JV^{2}M 2 el número de individuos depende *del ejercicio actual*, ya que debe existir al menos un individuo por cada recurso que se necesita para compilar el método particular.

Para poder realizar estas dos funciones, el objeto del juego de la manada consta de los siguientes componentes (figura 6.25):

- Un componente que hace que la posición del centro de la manada esté actualizada con la posición de la entidad: para eso, hemos creado un nuevo componente, `CPosCentroManada` que hereda de `CTransformable`. Cuando se recibe un mensaje del tipo `CSetPosition`, además de realizar las operaciones habituales, se avisa al gestor de la manada, para que cambie su centro. Ese centro afectará posteriormente al comportamiento de los individuos, gracias a la existencia de una regla que intenta mantenerlos cerca de él[28].

- Un componente que sigue la ruta de un grafo: en este caso el componente recibe en su inicialización el nombre del grafo y consigue su descripción preguntando al módulo correspondiente. Cuando el componente se activa hace que la posición de la entidad cambie, recorriendo las aristas del grafo. Por lo tanto, en cada vuelta del bucle emite el mensaje `CSetPosition` a todos sus componentes hermanos para que actúen en consecuencia. Cabe destacar que este componente es completamente *independiente* de la idea de manada, y por lo tanto puede utilizarse para controlar la posición de cualquier entidad, por ejemplo de un soldado patrullando.

- Un componente que configura la manada: al crearse la entidad para la IA de la manada, el propio objeto del juego *debe* registrar la manada

[28] Existe otra posibilidad de diseño distinta, que consiste en crear un componente independiente de `CTransformable` que, al recibir el mensaje `CUpdatePosition` cambie el centro de la manada; ambas opciones son válidas.

en el gestor. Para eso, se crea el componente `CConfigManadaRecursos` que es destruido inmediatamente después de que la inicialización se lleve a cabo. Sus funciones son varias:

- Crear la manada inicial en el gestor de manadas: de acuerdo a las propiedades establecidas en el propio mapa por el diseñador, crea la manada configurando las reglas que se utilizarán.
- Crear los individuos iniciales: es decir, el componente es el que implementa la configuración del entorno en base al ejercicio descrita en el apartado 6.4.3.En su inicialización, comprueba el método en ejecución en el *frame* activo y, en base a los recursos que se necesitan para su ejecución, crea los individuos de la manada que los llevan. La creación de los individuos se hace por partida doble: por un lado en el gestor de manadas se indica el número de individuos que hay que controlar, y por otro lado, se crean los objetos del juego que representan a cada uno de ellos. Los componentes con los que se configuran estos objetos del juego son tratados posteriormente.
- Activar el componente de seguimiento de ruta mediante un grafo: es decir, enviar el correspondiente mensaje para que la posición de la entidad siga ese grafo. La *activación explícita* del componente permite que otras entidades (como el NPC patrullando que nombrábamos antes) dispongan del mismo componente de seguimiento de aristas del grafo, que desactivan momentáneamente para realizar otra tarea distinta.

A la vista de esta lista de componentes, es interesante advertir que este objeto de juego *no* contiene un componente de representación gráfica ni de colisión, al contrario de los ejemplos nombrados hasta ahora.

6.4.7.3. Individuo de la manada

Cada uno de los individuos de la manada (que pueden verse en la figura 6.17) son modelos animados (actores) cuya posición viene determinada externamente por el controlador de la manada (`CManada`).

Cada uno de los individuos oculta un recurso necesario para la ejecución del método del *frame* en el que se encuentra. Además, el jugador puede *cazarlos* utilizando su arma. Cuando el individuo muere, el objeto que ocultaba pasa al inventario del jugador.

Los componentes con los que está configurado el `OIMObject` responsable de cada una de estas entidades son los siguientes (figura 6.26):

- Componente gráfico: en este caso, como es un avatar con animaciones, se utiliza un `CActorComponent`, ya que permite controlar las animaciones.

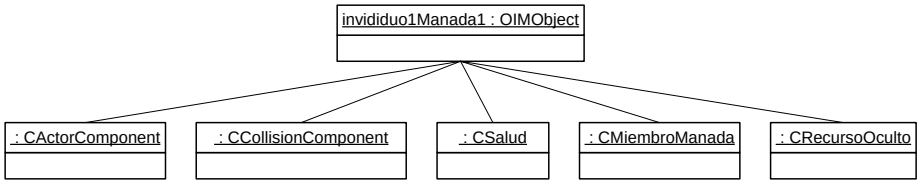


Figura 6.26: Componentes utilizados en los individuos de las manadas

- Componente de colisión: que instala una malla de colisión en el motor de colisiones, de tal forma que el estudiante no pueda *atravesar* al personaje. También se utiliza para que se detecten las colisiones con el disparo, de tal forma que se sepa cuándo el avatar está siendo *cazado*.

- Componente de nivel de vida: aporta un nuevo atributo a la entidad que es la *salud* o nivel de vida que tiene. Cuando recibe un mensaje de notificación de colisión con el disparo, el nivel baja, hasta que llega a cero. En ese momento, el componente envía un mensaje del tipo `CMuerto`.

- Componente de pertenencia a una manada: el avatar que aparece en el entorno *no* es el responsable de decidir en su posición y dirección, sino que ésto viene determinado por las propiedades de la manada (clase `CManada`). Debido a esto, el componente que fija la *transformación* (o posición) de la entidad *pregunta* en cada vuelta del bucle a su manada correspondiente en qué posición debe aparecer el individuo. El componente lo hemos implementado en la clase `CMiembroManada` que hereda de `CTransformable`. En cada actualización, comprueba la nueva posición y la envía con el mensaje del tipo `CUpdatePosition` para que el resto de componentes (en particular el gráfico y el de colisión) se actualicen[29].

  Una segunda función de este componente es detectar cuándo el personaje ha muerto (mensaje `CMuerto`). Cuando ésto ocurre, se lo comunica a la manada para que sea eliminando de ésta.

- Componente que gestiona el recurso oculto: hemos dicho que cada individuo oculta un recurso necesario para la ejecución del *frame* actual. Para eso, existe un componente que, ante un mensaje indicando que el personaje ha muerto, mete ese recurso en el inventario del jugador.

[29] Igual que en el caso de la inteligencia artificial de las manadas vista anteriormente, el diseño de clases podría haberse hecho de tal forma que `CMiembroManada` fuera independiente de `CTransformable`, de tal forma que el primero en cada vuelta del bucle envía un mensaje `CSetPosition`, que es traducido por el segundo a otro de `CUpdatePosition`.

Todos estos componentes son configurados en el momento de creación de la inteligencia artificial de la manada. En particular, son preestablecidos por la manada los siguientes valores: modelo del avatar (para todos los individuos el mismo modelo, que viene como información en el mapa del nivel), recurso que oculta (extraído de la información del método activo), nivel de salud (constante establecida en el mapa) y manada y número de individuo por el que preguntar la posición a la clase `CManada`.

6.5. Vista lógica

Recordemos, según lo visto en la sección 5.5 que en este módulo se almacena el estado del mundo a un nivel de abstracción lo suficientemente alto como para poder aplicar las técnicas de razonamiento adecuadas. También aquí situamos el comportamiento inteligente de los objetos del entorno más importantes desde el punto de vista del sistema educativo, así como la monitorización del estudiante.

En el caso de JV^{2}M, en la vista lógica es donde se encuentra la implementación de la máquina virtual de Java simplificada de acuerdo a lo indicado en la sección 6.2.3.

En cuanto a las otras dos funciones (comportamiento inteligente del algún objeto y monitorización del estudiante), tanto en JV^{2}M 1 como en JV^{2}M 2 consisten prácticamente en meros *puentes* desde el módulo OIM al subsistema de tutoría.

Con respecto al comportamiento inteligente de objetos del entorno importantes, únicamente en JV^{2}M 1 existe un "objeto" del entorno importante y relevante desde el punto de vista del sistema educativo: el agente pedagógico. En la vista lógica se sitúan lo que llamamos "módulo de percepción" y "módulo de movimiento". La primera de ellas recibe los eventos de alto nivel que ocurren en el entorno (el usuario quiere hablarnos, por ejemplo), y lo redirige al módulo cognitivo del agente situado en el subsistema de tutoría. El módulo de movimiento hace lo propio. Cuando recibe las órdenes a realizar por el agente, las divide en acciones primitivas en el entorno virtual, y las manda al avatar (entidad) en la capa inferior. En particular, algunas de las órdenes pueden ser directamente las *microinstrucciones* que hay que ejecutar en la máquina virtual; el módulo de movimiento *traduce* esas microinstrucciones en operaciones sobre el entorno ("usar X con Y"), y las envía.

La monitorización del estudiante por su parte recibe las acciones que éste realiza sobre el entorno (en concreto, sobre los objetos que representan estructuras de la máquina virtual), e informa de ellos al subsistema de tutoría. Éste actualiza el progreso del ejercicio y toma las decisiones que estima oportunas.

En cuanto a la implementación de la máquina virtual, ésta cubre sus

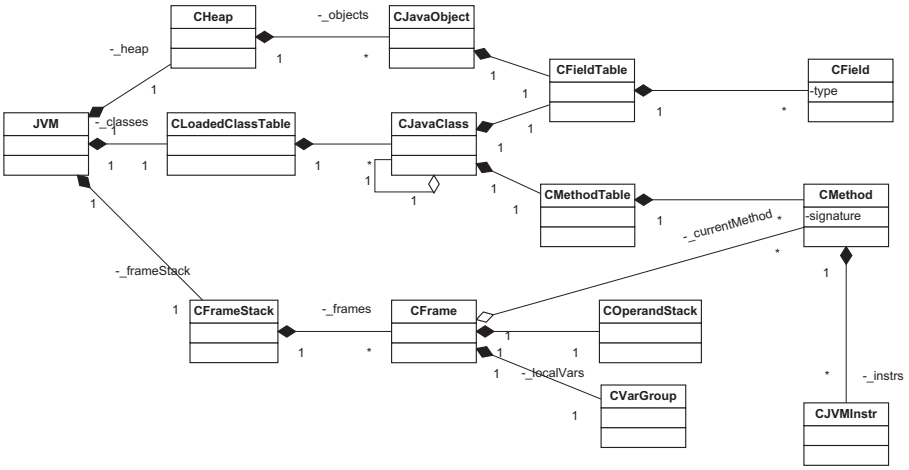


Figura 6.27: Diseño de clases de la implementación de la JVM en JV^{2}M

estructuras básicas. Dispone de una lista de clases cargadas, una lista de objetos creados y una pila de *frames*. Las clases tienen una lista de métodos (con sus instrucciones máquina asociadas), y una tabla con todos los campos de la clase (tanto estáticos o de clase, de los que se almacena también su valor, como de instancia, de los que únicamente se almacena su nombre y tipo). Los objetos contienen una referencia a la clase a la que pertenecen, y una tabla de sus campos con los valores. Los *frames* contienen una referencia al método que están ejecutando, así como una pila de operandos y un conjunto de variables locales. El diseño de clases utilizado en JV^{2}M aparece en la figura 6.27.

La implementación incluye un cargador de clases especial, que interpreta el fichero que contiene el ejercicio que debe resolver el usuario. En ese fichero aparecen todas las clases necesarias, con las que el cargador inicializa la máquina virtual.

Lo más relevante, no obstante, de la implementación es que gran parte de las clases que representan estructuras de la JVM permiten *observadores*, de tal forma que elementos externos a la propia implementación pueden ser informados sobre cambios en su estructura interna. Así, por ejemplo la pila de operandos (cuya implementación aparece en la sección C.3.1) informa cuando se apila o desapila algún valor, los campos avisan cuando sus valores se alteran, el heap emite un mensaje cuando algún objeto es creado, y las variables locales informan cuando se crean nuevas o cambian los valores de las antiguas.

De esta forma, los objetos del juego o entidades del módulo OIM (sección 6.4) son avisadas cuando ocurre alguno de estos cambios para que alteren la representación de esa estructura particular en el entorno virtual. La ta-

Clase	Descripción
`CHeap`	Informa de la creación de nuevos objetos.
`CLoadedClassTable`	Informa ante la carga de nuevas clases.
`CFrameStack`	Informa ante la creación o destrucción de un nuevo *frame*.
`COperandStack`	Informa cuando en la pila de operandos se apila o desapila un nuevo valor.
`CVarGroup`	Informa cuando algún valor en el grupo de variables locales es alterado.
`CField`	Informa cuando un campo (de clase o de objeto) es modificado externamente.

Tabla 6.2: Clases de la JVM que permiten observadores

bla 6.2 resume cuáles de las clases de la figura 6.27 notifican sus cambios de estado a terceros.

6.5.1. Ejecución de instrucciones de la JVM en JV^{2}M 1

La metáfora de JV^{2}M 1 exige que el estudiante *ejecute* por él mismo las instrucciones de la máquina virtual, manipulando los objetos del entorno. En este sentido, existía el concepto de *microinstrucciones*, que son todos aquellos *pasos atómicos* que los avatares pueden realizar sobre él.

Por tanto, el estudiante debe manipular cajas (datos) y hablar con personajes para que se ejecuten cada una de las microinstrucciones en la máquina virtual, encaminadas a la ejecución completa de una instrucción máquina. Algunas instrucciones se componen únicamente de una microinstrucción (por ejemplo, apilar un valor constante), mientras que otras pueden involucrar numerosas instrucciones (por ejemplo, la invocación a un método).

Independientemente del número de pasos, lo que debe quedar claro es que es *el propio estudiante* (o en su defecto el agente pedagógico) quien *ejecuta* las instrucciones máquina. Por lo tanto, en la práctica, la implementación de la máquina virtual en JV^{2}M 1 *no* requiere la capacidad de ejecución de las instrucciones, ya que eso se hace "desde fuera". Exige, eso sí, que ciertos métodos que en condiciones normales no serían accesibles sean públicos. Esto es necesario para que desde fuera de la implementación sea posible ejecutar una a una esas microinstrucciones, accediendo a sus estructuras internas.

6.5.2. Ejecución de instrucciones de la JVM en JV^{2}M 2

En el diseño de JV^{2}M 2, liberamos al estudiante de la ejecución de cada uno de los pasos primitivos o atómicos. En vez de hacer eso, simplemente debe *teclear* en el terminal del entorno el mnemónico de la instrucción máquina que desea ejecutar (ver figura 6.4a).

Debido a esto, ya no es el estudiante (o el agente pedagógico) el que debe conocer qué estructuras deben alterarse en la JVM para implementar cada una de esas instrucciones, sino que ese conocimiento debe estar codificado en este módulo, la *vista lógica*.

Por lo tanto, en JV^{2}M 2, se ha creado un "*ejecutor*" de las instrucciones de la máquina virtual (`JVMExecutor`), dentro de la vista lógica. En particular, dentro de los tres tipos de responsabilidades de este módulo indicados en la sección 5.5[30], la ejecución de las instrucciones recae en la inteligencia artificial o comportamiento de alto nivel de la entidad que representa al terminal en el entorno virtual.

Cuando el terminal detecta que está siendo utilizado por el estudiante, lo que éste escribe es redirigido a la vista lógica para su procesado.

6.6. Subsistema de tutoría

El funcionamiento interno del subsistema de tutoría de JV^{2}M no forma parte de este trabajo, sino que es ampliamente descrito por Gómez Martín (2007). En esta sección aparece una breve descripción, que trata únicamente los aspectos relevantes desde el punto de vista arquitectónico.

6.6.1. Inicialización

Cuando la aplicación se lanza, el subsistema recibe momentáneamente el control para que pueda ejecutar sus funciones de inicialización (igual que ocurre con el resto de módulos). En ese momento el subsistema carga ficheros externos que contienen el conocimiento dependiente del dominio pero independiente del estudiante particular que utilizará el sistema. También es en ese momento cuando se inicializan los gestores de los ejercicios y de los perfiles de usuario que aparecían en la figura 5.2.

En ambas versiones de JV^{2}M se carga en este momento una ontología sobre los conceptos que se deben enseñar, que incluye tanto las estructuras de la JVM como los conceptos de compilación del lenguaje Java (figura 4.9). Por su parte, el gestor de ejercicios comprueba cuántos hay disponibles para poder después seleccionarlos en base a las estrategias pedagógicas.

Además de esa información, cada una de las dos versiones de JV^{2}M necesitan información específica:

- **JV^{2}M 1**: debido a la existencia de JAVY, el agente pedagógico, y al modo de ejecución de las instrucciones el subsistema de tutoría de JV^{2}M 1 necesita un tipo de conocimiento concreto no necesario en la versión posterior. En particular JAVY necesita conocer las microinstrucciones

[30] Recordemos que eran (i) simulación, (ii) inteligencia artificial de alto nivel y (iii) monitorización del estudiante.

necesarias para ejecutar cada una de las instrucciones máquina. También se cargan las explicaciones que el agente proporciona cuando se mantiene una conversación con él.

- **JV^{2}M 2**: en este caso, la ejecución de las instrucciones máquina no son desglosadas en varias etapas, por lo que no es necesario un conocimiento tan concreto. Sin embargo, en esta versión se ha incluido una ayuda sintáctica y semántica que permite a los jugadores lanzar preguntas en lenguaje natural. Para la implementación de esta ayuda, se ha utilizado jCOLIBRI (Recio-García et al., 2005) un framework implementado en Java para la creación de sistemas CBR (*Case-based reasoning*, Razonamiento basado en casos); en particular, se ha utilizado la extensión para CBR textual (Recio, Díaz Agudo, Gómez Martín y Wiratunga, 2005). Dado que jCOLIBRI está implementado en Java, el subsistema de tutoría en su inicialización *lanza* la máquina virtual de Java[31]. Para evitar alargar la inicialización completa de la aplicación, se utiliza una hebra secundaria que se lanza en el momento de la inicialización, de forma que la aplicación puede empezar a ejecutarse (presentando el menú de opciones, por ejemplo) antes de terminar la lectura completa de todas las ontologías de términos e indexado de textos necesarios. Los detalles exactos de implementación aparecen en el apéndice B.

La inicialización, no obstante, no está completa hasta que el usuario no se identifica en el sistema. Al arrancar la aplicación, ésta presenta un menú al jugador que le permite crear un nuevo perfil, o seleccionar uno ya existente (ver figura 6.1). Ese perfil se redirige hacia el subsistema de tutoría utilizando la monitorización del estudiante de la vista lógica. El subsistema de tutoría avisa al gestor de perfiles de la selección, para que cargue el perfil concreto. Ese perfil será posteriormente utilizado para seleccionar el siguiente ejercicio a presentar al usuario.

6.6.2. Ejercicios

Cuando el usuario establece el perfil, el subsistema de tutoría decide qué conceptos debe poner en práctica el estudiante en el siguiente ejercicio utilizando la estrategia pedagógica que considere oportuna. Con esos conceptos, formula una consulta al gestor de ejercicios, que recupera el más parecido de su base de ejercicios.

La selección del ejercicio será la que posteriormente determine el aspecto final que tendrá el entorno virtual en el que el usuario se ve inmerso. Esto es así gracias a la metodología de creación de contenidos que describíamos en el capítulo 4.

[31] No confundir esta JVM con la implementada en la vista lógica del sistema. La JVM a la que nos referimos aquí es la implementada por Sun, que puede utilizarse desde una aplicación en C++.

En JV^{2}M, los ejercicios están almacenados en ficheros XML, que contienen tanto el código Java como el código compilado para la máquina virtual. Aunque escapa al ámbito de este trabajo, indicar que esos ejercicios también tienen explicaciones y referencias a otras ontologías externas sobre el dominio, que son aprovechadas por el módulo pedagógico y módulo experto.

El proceso de inicialización comienza cuando el subsistema de tutoría selecciona el siguiente ejercicio. En ese momento, inicializa la vista lógica del sistema de acuerdo a él. En nuestro caso, esta inicialización consiste, precisamente, en hacer que el cargador de clases de la JVM implementada en la vista lógica *cargue* las clases que se necesitan en el ejercicio. Una vez cargadas, el subsistema también *establece* el estado inicial de la máquina virtual. Así, si el ejercicio comienza la ejecución en la función `main` de una determinada clase, el subsistema también crea el *frame* para la ejecución de ese método. Si el ejercicio comienza en otro punto del programa, lanza la ejecución de éste hasta que se llega a ese punto.

Una vez configurado, el motor de la aplicación comienza la carga de un mapa (entorno virtual), al principio de la partida. En ese momento, *avisa* a la gestión de entidades del comienzo inminente en la carga de un mapa, y procede a la creación de los objetos del juego. Éstos basarán su inicialización en el estado de la misma, como ya hemos descrito en la sección 6.4.3.

6.6.3. Ejecución durante la partida

Durante el juego el subsistema de tutoría también debe ejecutarse cada cierto tiempo para mantener actualizada toda su información y tomar las decisiones que considere oportunas en base a los acontecimiento que suceden en el entorno virtual.

Esta ejecución en JV^{2}M se realiza de dos formas distintas:

- En algunos momentos, la vista lógica *invoca* al subsistema de tutoría, avisándole de los eventos relevantes. En la figura 6.28 aparecen tres ejemplos de invocaciones directas:
  - El simulador o implementación de la máquina virtual informa al sistema experto de la ejecución de una instrucción o microinstrucción. El sistema experto entonces es capaz de comprobar si el estudiante está actuando correcta o incorrectamente.
  - El avatar de JAVY que representa al agente pedagógico en el entorno virtual (en JV^{2}M 1), informa al subsistema de tutoría que el estudiante desea establecer una conversación con él.
  - El encargado de la monitorización del estudiante indica que éste lleva inactivo demasiado tiempo, o que se ha seleccionado un perfil nuevo (invocado cuando la aplicación está en el menú).

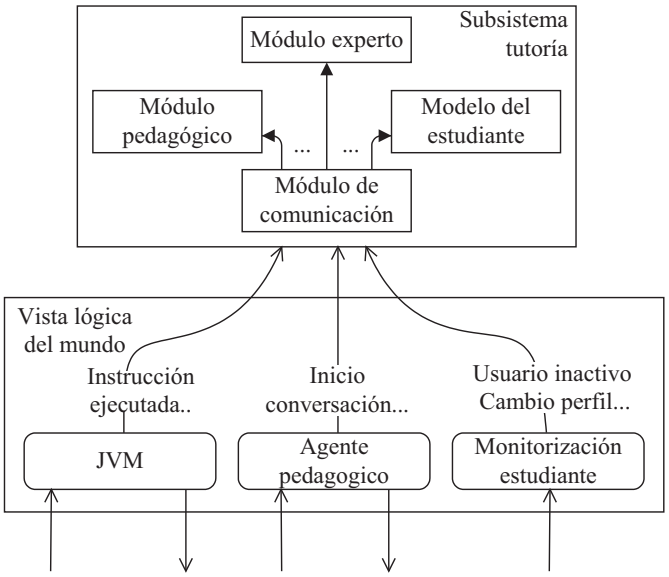


Figura 6.28: Ejemplos de invocaciones de la vista lógica al subsistema de tutoría

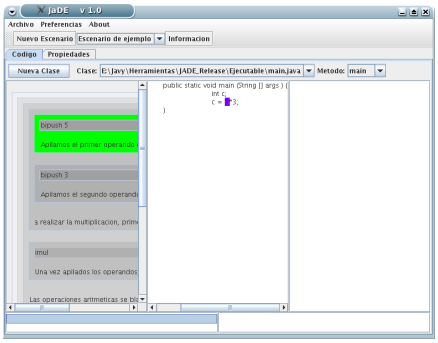
- El subsistema de tutoría para no depender de la vista lógica en cuanto al control de ejecución también dispone de una hebra independiente. Cuando el subsistema es inicializado, éste lanza una nueva hebra de larga duración que será la responsable de las decisiones del subsistema cuyo tiempo de respuesta sea largo. Así, los módulos del subsistema involucrados pueden ejecutarse concurrentemente al resto de la aplicación, sin bloquear ésta. En JV^{2}M, la hebra se encarga de las decisiones de alto nivel del comportamiento del agente pedagógico entre otras cosas.

6.7. Generación de contenidos

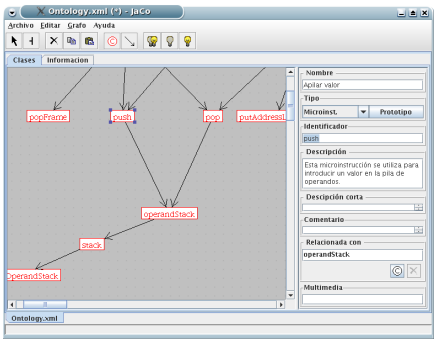
La arquitectura para el desarrollo de aplicaciones educativas presentada en el capítulo 5 es una arquitectura *dirigida por datos*. Recordemos de la sección 2.4 que este tipo de organización elimina la mayoría de los datos utilizados por la aplicación del fichero ejecutable, para almacenarlos en ficheros externos.

La metodología para la creación de contenidos dirigidos por datos que veíamos en el capítulo 4 se basaba en la clara división en dos partes de esos datos: por una lado aquellos relacionados con el conocimiento específico del dominio, y por otro lado la generación de los niveles o del entorno virtual.

En los dos apartados siguientes describimos cómo se ha realizado la crea-



(a) Herramienta de edición de ejercicios para JV^{2}M



(b) Herramienta de edición de ontologías para JV^{2}M

Figura 6.29: Capturas de dos herramientas de generación de conocimiento del dominio en JV^{2}M

ción de ambos contenidos en JV^{2}M.

6.7.1. Generación del conocimiento del dominio

Este conocimiento es específico del dominio que se está enseñando y de la forma en la que funciona el subsistema de tutoría. Cuando describíamos el módulo utilizado en JV^{2}M en la sección 6.6, indicábamos que su funcionamiento interno no pertenecen a este trabajo, sino que son descritos por Gómez Martín (2007). De forma similar, las características exactas de los datos que se precisan para expresar el conocimiento enseñado en JV^{2}M, también pertenece a ese trabajo.

Desde el punto de vista arquitectónico, lo único importante es que todos esos datos se almacenan en ficheros externos al propio ejecutable, lo que permite la reutilización de parte del módulo.

Para la generación de todo ese conocimiento, se desarrollaron diversas herramientas, que permiten la creación tanto de los ejercicios Java (figura 6.29a) como de las ontologías que el subsistema de tutoría utiliza (figura 6.29b).

6.7.2. Generación del entorno de aprendizaje

El entorno virtual en el que suceden los episodios de aprendizaje está formado principalmente por dos tipos de ficheros de datos: (i) los modelos animados de los personajes y demás entidades y (ii) los mapas que contienen la información de las entidades y el modelo de las estructuras estáticas del entorno.

Dejando de un lado los modelos que utilizamos en la versión preliminar de JV^{2}M con WildMagic (figura 6.6), el resto de modelos de JV^{2}M han sido creados utilizando Alias|Wavefront Maya, un software para modelado 3D que

ha sido utilizado en la creación de numerosas películas y juegos.

Para poder cargar los modelos generados en Maya en el motor gráfico utilizado, hemos hecho uso de sendos plug-ins:

- **JV^{2}M 1**: recordemos que esta versión utiliza Nebula 1 y modelos con el formato de Half-Life. En este caso, realizamos una modificación[32] de un plug-in desarrollado por Reality Factory[33] que permite grabar a un tipo de archivo especial que finalmente se utiliza para crear el fichero que carga el motor gráfico.
- **JV^{2}M 2**: en este caso, Nebula 2 dispone de un plug-in comercial para Maya que permite exportar directamente los modelos a los ficheros nativos del motor (ver figura 6.8).

La existencia de estos plug-in mejora la independencia entre el motor gráfico utilizado por la aplicación y la herramienta de generación de los contenidos. Para ambas versiones se utilizó la misma herramienta de modelado, pero utilizando distintos exportadores. Por tanto, si se desea cambiar el motor gráfico conservando los modelos, únicamente se requiere la existencia de exportadores en Maya para él. Si el nuevo exportador no impone restricciones excesivas en las características del modelo (como configuración del esqueleto o propiedades de las texturas) los modelos creados en el programa de modelado podrán ser exportados al nuevo motor fácilmente. De hecho, aunque estamos diciendo continuamente que JV^{2}M 1 utiliza Nebula 1, podemos decir que también existió temporalmente una versión con Nebula 2 como motor gráfico que presentaba los modelos de JV^{2}M 1 exportados para Nebula 2.

Por su parte, el formato de los ficheros de mapa es impuesto por el motor de la aplicación[34]. Por ejemplo, el motor del juego de Far Cry exige mapas creados con su editor SandBox (Crytek, 2004), el motor de Unreal Tournament (Epic Games, 1999) utiliza los mapas generados con UnrealEd, mientras que la saga de Half-Life (Valve Software, 1998) es capaz de interpretar mapas creados con Worldcraft o Hammer.

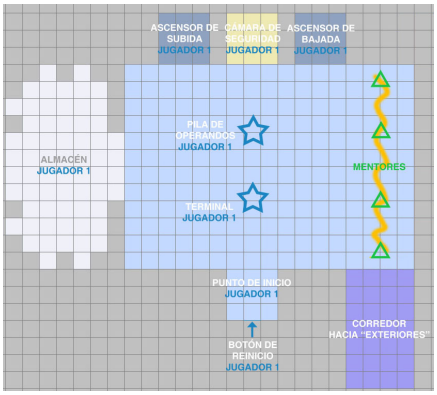
Esto hace que la sustitución de un motor de aplicación por otro suponga no solo cambios en el código sino la regeneración completa del entorno de aprendizaje, debido a que los formatos de los editores son incompatibles entre sí.

En nuestro caso, la implementación del motor de la aplicación es propia, por lo que el formato en el que leemos los mapas no viene impuesto desde

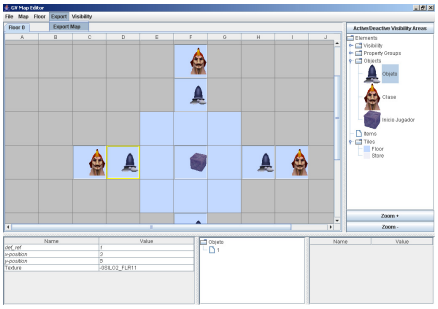
[32] Disponible en `http://gaia.fdi.ucm.es/grupo/projects/javy/devzone.html`

[33] `http://www.realityfactory.info`

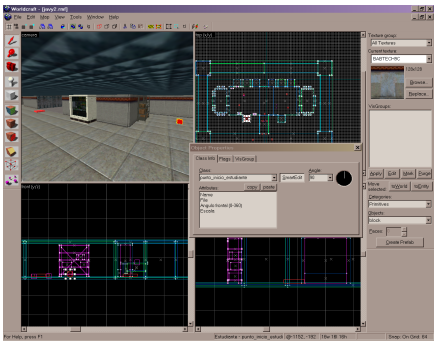
[34] Esto es especialmente cierto si el código fuente del motor no está disponible; si éste es abierto, se podrá incluir soporte para otros formatos, aunque en algunos casos el motor gráfico puede imponer unas limitaciones que hagan imposible el uso de algunos de ellos.



(a) Boceto del mapa principal de JV^{2}M 2



(b) Captura del editor GV Map Editor usado en JV^{2}M 2



(c) Captura de Worldcraft utilizado en JV^{2}M 2

Figura 6.30: Generación de mapas en JV^{2}M

fuera. Ya explicamos en la sección 6.3.3 que el cargador de los mapas es lo suficientemente general como para poder soportar distintos formatos de forma transparente al resto de la aplicación.

A pesar de esta versatilidad en cuanto a la poca imposición en los formatos de los mapas, la creación de los mapas de JV^{2}M ha sido realizada intentando minimizar el impacto que tendría la sustitución de nuestro motor de la aplicación por otro distinto.

En particular, el proceso de creación de los mapas en JV^{2}M 2 ha utilizado dos editores distintos. El primero de ellos es independiente del formato final de los mapas, y permite realizar la versión inicial. Mediante un conversor, entonces, se graba en un formato directamente interpretable por un segundo editor con el que se generará el mapa. Ese segundo editor es utilizado para refinar el mapa y darle el aspecto final. El fichero generado será por último interpretado por el motor de la aplicación. De esta forma, si el motor de juego actual es sustituido por otro, la versión inicial del mapa puede ser reutilizada.

Con este planteamiento, las etapas por las que se ha pasado para la creación de los entornos en JV^{2}M 2 son las siguientes:

- Diseño sobre papel del mapa: este boceto aparece en el documento de diseño del juego (figura 6.30a). Contiene una idea del mapa y de la colocación de las entidades más importantes.

- Creación de la versión inicial del mapa: utilizando el boceto, se genera una primera versión del mapa. Dado que JV^{2}M 2 se desarrolla en interiores, utilizamos un editor basado en cuadrícula llamado GV Map Editor[35] (ver figura 6.30b), que facilita la creación de los muros y demás componentes arquitectónicos. El editor también permite colocar las entidades fundamentales del nivel.

- Conversión a formato específico: en este caso, el propio editor permitía grabar directamente en el formato soportado por el segundo editor.

- Creación de la versión definitiva del mapa: utilizando en este caso Worldcraft (figura 6.30c), que finalmente graba los ficheros en el formato `bsp` utilizado. De esta forma, JV^{2}M permitiría de forma fácil remplazar el motor de la aplicación por el utilizado en Half-Life, que soporta estos mapas de manera nativa.

Resumen

En este capítulo hemos descrito la implementación de la aplicación más importante desarrollada con la metodología y arquitectura descritas en los capítulos 4 y 5 respectivamente.

En particular, el capítulo comenzaba con una breve descripción de JV^{2}M 1 y JV^{2}M 2, las dos versiones de una aplicación para el aprendizaje de las estructuras de la máquina virtual de Java y la forma en la que un programa escrito en Java es compilado a código máquina de la misma.

Posteriormente, se ha hecho un repaso a cada uno de los módulos de la arquitectura que describíamos de forma general en el capítulo 5, detallando la implementación concreta utilizada.

Finalmente, se ha descrito la forma en la que hemos generado los contenidos utilizados en JV^{2}M, utilizando la metodología de creación de contenidos descrita también de forma general en el capítulo 4.

[35] Este editor fue desarrollado por los alumnos del Máster de Videojuegos de la Universidad Complutense de Madrid durante el curso 2005-2006.

Notas bibliográficas

La metáfora utilizada en JV^{2}M 1 que veíamos en la sección 6.2.1 ha sido descrita en Gómez Martín, Gómez Martín y González Calero (2005d) y Gómez Martín, Gómez Martín y González Calero (2006a). Por su parte, JV^{2}M 2 es descrito en Gómez Martín, Gómez Martín, Palmier Campos y González Calero (2006c). Por último, una comparativa entre ambas versiones, que destaca la importancia de una buena selección de metáforas durante la fase de diseño es Gómez Martín, Gómez Martín, González Calero y Palmier Campos (2007d).

Con respecto a la arquitectura, ha sido descrita, aunque no de forma tan exhaustiva como en este capítulo, en Gómez Martín, Gómez Martín y González Calero (2007a) y Gómez Martín, Gómez Martín y González Calero (2007b).

Una variación de la arquitectura es descrita en Peinado Gil, Gómez Martín y Gómez Martín (2005), aunque en este caso su uso no es para la creación de videojuegos educativos, sino para la implementación de programas de entretenimiento con una fuerte componente narrativa.

Por último, la descripción completa de la forma en la que está implementado el subsistema de tutoría de JV^{2}M forma parte del trabajo de Gómez Martín (2007).

Capítulo 7

Conclusiones y trabajo futuro

Quien crea que llegué al puerto definitivo es que no comprende el espíritu del verdadero homo viator.

Pedro Jesús Fernández, Peón de Rey

A lo largo de los capítulos de esta memoria hemos descrito una arquitectura y metodología para el desarrollo de aplicaciones educativas basadas en videojuegos. En este último capítulo pasamos a describir las que consideramos aportaciones más importantes de este trabajo, así como las posibles líneas de continuación y trabajo futuro.

7.1. Conclusiones

El trabajo aquí presentado se enmarca dentro de las aplicaciones educativas. Dentro de este amplio campo encontramos tanto aquellas aplicaciones de enseñanza tradicionales, basadas en la mera presentación de contenido multimedia, como los sistemas educativos basados en videojuegos donde el usuario-estudiante *juega* un papel activo dentro de su propio aprendizaje que se adorna con contenidos lúdicos que incrementan la motivación.

El primer tipo de aplicaciones, más sencillo, se ha utilizado ampliamente durante las últimas décadas. El florecimiento de estos sistemas ha sido posible gracias a su simplicidad. En la mayoría de los casos, la interacción entre el usuario y el programa está limitada a unos pocos mecanismos, igual que lo está la forma en la que la aplicación puede presentar los contenidos. Gracias a esta acotación de las posibilidades, durante el proceso de autoría se dispone de un conjunto de operaciones establecidas de los que no se puede salir. De esta forma, ha sido posible la construcción de *editores* de generación de contenidos cuyo producto final (el conjunto de ficheros con los objetos de aprendizaje) es mostrado por un *visor* genérico. Con la mera ejecución de

ese visor configurado para interpretar el conjunto de contenidos, se dispone de un sistema que permite a un usuario interactuar con la colección de datos para aprender los conceptos del dominio deseados.

Esta clara división entre los contenidos o datos que utiliza el programa y el propio visor aparece también en otros campos. En general, se puede decir que esta separación es posible en dominios consolidados donde existe mucha experiencia previa de desarrollo de aplicaciones no dirigidas por datos. Gracias a esa experiencia, se han podido establecer los límites de lo que se permite hacer al usuario y lo que no, es decir, el conjunto de operaciones que puede realizar, así como las herramientas que tiene disponibles el diseñador para variar los contenidos ofrecidos. Por nombrar algún ejemplo distinto al ya visto, podemos recordar las aplicaciones de *ficción interactiva* que presentaban una historia de ficción (inicialmente de forma textual) donde el usuario podía introducir órdenes para hacerla avanzar; rápidamente se hizo un análisis de las mismas, se acotó el tipo de órdenes que se podían aceptar y se crearon lenguajes para generación de las historias interactivas que eran interpretados por los *motores* correspondientes.

El principal problema cuando se extrapola este modelo de creación de contenidos a aplicaciones educativas basadas en videojuegos es que la heterogeneidad de los mismos no permite establecer claramente las primitivas de variabilidad. Esto es debido a que en las aplicaciones existen dos tipos de contenidos: por un lado existen los contenidos pedagógicos y por otro lado, los entornos virtuales que determinan la jugabilidad. A la dificultad de determinar los ejes de variabilidad de estos contenidos tan heterogéneos, se le añade la necesidad de actuación de expertos distintos: aquellos que generan el contenido educativo (expertos en el dominio que se enseña) y los que generan los entornos virtuales (diseñadores de niveles).

La principal aportación de esta tesis consiste en proponer una solución para este problema de generación de aplicaciones educativas basadas en videojuegos, a través de dos ideas fundamentales:

- La definición de una metodología de creación de contenidos donde se separa claramente el trabajo realizado por ambos expertos.
- La creación de una arquitectura que permite el desarrollo de estas aplicaciones donde el contenido ha sido generado por separado.

Estas ideas se concretan en una serie de resultados que constituyen las aportaciones de nuestro trabajo:

- Se ha estudiado el proceso de creación de videojuegos tradicionales, detectando los problemas más importantes en cuanto a arquitectura y generación de contenidos se refiere.

- Se ha analizado la naturaleza de las aplicaciones educativas, y sus métodos de construcción y generación de contenidos.
- Se ha identificado el problema de la creación de contenidos en las aplicaciones que aunan ambas áreas, las aplicaciones educativas basadas en videojuegos.
- Se ha definido una metodología para la creación de estas aplicaciones, definiendo las etapas de desarrollo, el tipo de profesionales que deben participar, y los tipos de contenidos que deben generar.
- La metodología resuelve el problema de interacción entre los distintos profesionales, permitiendo la creación de contenidos por separado.
- La separación de ambos tipos de datos permite la reutilización de los mismos, con la reducción de costes que ello supone. Por ejemplo, el mismo entorno virtual puede utilizarse en distintos ejercicios.
- Hemos visto como la existencia separada de entorno y ejercicios permite también el uso de distintas técnicas de extracción de información sobre ellos para analizar sus propiedades e identificar y resolver posibles problemas en los mismos.
- Se ha comprobado que el análisis formal de conceptos puede ser utilizado para comprobar la cobertura del conjunto de ejercicios creados por el experto del dominio.
- Hemos descrito un nuevo uso del análisis formal de conceptos para el análisis de las partidas de un juego, ayudando así a los creadores de los mapas o entornos virtuales en el momento de *nivelar* la dificultad del juego.
- Se ha definido una arquitectura software que soporta la metodología expuesta.
- Hemos descrito como esta arquitectura permite configurar el comportamiento de la aplicación en función del entorno virtual y de los ejercicios presentados.
- Se ha mostrado la aplicabilidad de la metodología y arquitectura por medio de diversas aplicaciones educativas.
- Se ha detallado la instanciación de la arquitectura que ha dado lugar a dos aplicaciones educativas para el aprendizaje de la máquina virtual de Java y cómo el código escrito en ese lenguaje es compilado para ella, JV^{2}M 1 y JV^{2}M 2.

- Utilizando el contexto de la aplicación anterior, hemos descrito cómo el uso de *componentes* para la programación de las entidades del juego facilita considerablemente el desarrollo de la aplicación y la reutilización de parte del código en otras aplicaciones.

- Por último, también hemos utilizado JV^{2}M para detallar un ejemplo real de configuración del entorno virtual en base a los ejercicios.

7.2. Trabajo futuro

Para finalizar esta memoria, vamos a comentar las líneas de continuación más interesantes, desde nuestro punto de vista, del trabajo aquí presentado.

En primer lugar, la separación del contenido en contenido pedagógico y jugabilidad puede suponer problemas si no se maneja con cautela. A pesar de que la metodología y arquitectura permiten una separación completa de ambos tipos de datos, es cierto que existen algunas dependencias entre ellos. Por ejemplo, es necesaria la existencia de unas entidades concretas en el entorno para poder configurar el ejercicio. Por tanto, debe garantizarse que la comunicación entre el equipo de creación de contenidos de instrucción y los diseñadores de los niveles están perfectamente coordinados. Para esta coordinación hemos propuesto el uso de herramientas de trabajo colaborativo, como los sistemas de control de versiones y de gestión del conocimiento. Sin embargo, prevemos una futura línea de trabajo en sistemas automáticos de mantenimiento de la *coherencia*. El sistema trabajaría con descripciones de alto nivel de las restricciones o exigencias de los ejercicios y de las capacidades que ofrecen los distintos entornos virtuales creados por los diseñadores. De esta forma, en tiempo de ejecución se podría garantizar la correcta configuración de los escenarios.

La existencia de estas descripciones de las propiedades de los entornos virtuales y de los ejercicios permitiría también la existencia de una colección de entornos distintos. Utilizando un mecanismo deductivo se pueden aplicar búsquedas sobre esa colección. En ese caso, el módulo que dado un ejercicio devuelve el entorno virtual que debe utilizarse que comentábamos en la página 150, utiliza esa búsqueda inteligente para obtenerlo. El módulo podría utilizarse para que un mismo ejercicio se resolviera en distintos entornos.

Otra posible línea de actuación es facilitar la construcción de las entidades del juego. La utilización de una arquitectura basada en componentes permite la definición de objetos de juego en ficheros externos, de tal forma que no es necesario regenerar el ejecutable de la aplicación para disponer de nuevos objetos. Sin embargo, y siguiendo con la misma idea que antes, es posible hacer una descripción de los *servicios* ofrecidos por cada componente, de tal forma que se pueden utilizar mecanismos deductivos y/o de planificación para construir, a partir de la descripción de comportamiento de una entidad,

qué componentes deben agruparse y cómo deben configurarse.

Por último, la división de los contenidos en dos ficheros independientes nos ha permitido la aplicación de una técnica como el análisis formal de conceptos para extraer propiedades interesantes de ellas. Sin embargo, creemos que esa línea está solamente esbozada y que existen muchas posibilidades de ampliación, para sacar aún más conclusiones útiles.

Apéndice A

Máquina virtual de Java

Este apéndice presenta una breve descripción del funcionamiento de la máquina virtual de Java (JVM). Es un pequeño resumen de la especificación completa de Sun, y se ofrece aquí para poder entender el capítulo 6, dedicado a JV^{2}M, una aplicación educativa sobre la estructura interna de JVM y la forma en la que Java es compilado.

A.1. Introducción

Java (Gosling et al., 2005) es un lenguaje de propósito general, orientado a objetos y concurrente. Su sintaxis es similar a la de C y C++, aunque elimina muchas de las características que hacen a éstos complejos. Sin embargo el éxito de Java no se debe únicamente a esa simplicidad, sino a su portabilidad.

El código compilado de Java no depende de ninguna arquitectura. En un tiempo donde las redes de ordenadores se han popularizado tanto, esta característica es muy apreciable. Un programa hecho en Java se puede ejecutar en cualquier máquina que disponga de un intérprete de Java.

En realidad, podemos decir que no es un intérprete en el sentido clásico de los intérpretes, donde se analiza el código fuente o una representación abreviada mediante tokens del mismo, y se ejecuta línea a línea. El código fuente de Java *se compila*. Pero el código objeto obtenido no son instrucciones para una arquitectura Intel/Windows, Solaris/Unix o Alpha/Linux, sino para una máquina virtual que se especificó a la vez que el propio lenguaje, y que es conocida como la Máquina Virtual de Java (Java Virtual Machine o JVM). Las características de la JVM han sido establecidas por Sun Microsystems (Lindholm y Yellin, 1999).

Los creadores de Java, además, han hecho un esfuerzo por crear un conjunto de clases que forman las *librerías* de Java que un programador puede asumir que estarán disponibles en todos los entornos donde se ejecute el programa. Por lo tanto, para que un programa Java pueda ejecutarse en

una arquitectura determinada, es necesaria la implementación de la JVM y del conjunto de estas clases, algunas de ellas implementadas utilizando los servicios propios del sistema operativo y máquina en cuestión.

A.2. Estructura de la máquina virtual

Al igual que una máquina real, la máquina virtual de Java está dividida en varias partes. Tiene una zona de memoria para albergar objetos dinámicos, una pila con los entornos de ejecución de los métodos, etc. En la especificación (Lindholm y Yellin, 1999), se detallan cada una de estas partes, las instrucciones que es capaz de ejecutar y en qué cambia cada una de ellas el estado de la máquina. Es decir, lo que se define es, a parte del formato del fichero donde se guarda el código Java compilado, el conjunto de instrucciones que posee la máquina y su cometido. Y eso es lo que debe ser capaz de realizar una implementación: leer los ficheros e implementar exactamente la semántica del código para la JVM que contiene.

Sin embargo, no hay restricciones a la hora de decidir cómo implementar una JVM. Una implementación podría coger la especificación literalmente, y para cada estructura de datos vislumbrada, construir un tipo de datos o una clase que la represente. También podría traducir, en tiempo de carga de cada clase, el código de cada instrucción en instrucciones nativas de la CPU donde se va a ejecutar (conocido como *just-in-time code generation*, o JIT).

A.2.1. Tipos de datos

Igual que en el lenguaje, en la máquina virtual existen dos tipos de datos: tipos primitivos y referencias, que contendrán, como su nombre indica, valores primitivos y referencias respectivamente.

Aunque el lenguaje Java es fuertemente tipado, la máquina virtual espera que casi todas las comprobaciones de tipos se hagan antes de comenzar a ejecutar, por el compilador primero y por la propia máquina virtual en tiempo de carga de las clases después. Por eso en las implementaciones, las variables de tipos primitivos no están marcadas con el tipo que tienen. Cuando la máquina va a ejecutar una instrucción que suma dos enteros, coge los valores de la memoria y hace la suma, sobreentendiendo que esos dos valores referenciados en los operandos eran en realidad enteros.

Los tipos primitivos son los mismos y tienen los mismos rangos de valores que en el lenguaje Java. No obstante, aunque el tipo `boolean` existe, se da poco soporte para él: no existe ninguna instrucción en la JVM que lo utilice. Las expresiones son compiladas utilizando instrucciones asociadas al tipo entero, donde el valor 0 representa `false`. Existe otro tipo primitivo que no tiene correspondencia con ningún tipo en el lenguaje, pero que es utilizado en tres instrucciones máquina, el llamado `returnAddress` que representa un

puntero al *opcode* de una instrucción.

Las referencias, igual que en el lenguaje, pueden serlo a objetos, a *arrays* o a interfaces. Sus valores son referencias a objetos creados dinámicamente o a `null`. Pueden verse como punteros a las estructuras de datos que guardan información de los objetos. Aunque la JVM no necesita guardar para cada referencia de qué tipo es, sí necesita saber a qué clase pertenece el objeto al que apunta.

A.2.2. Áreas de datos en tiempo de ejecución

Estas zonas de datos se utilizan durante la ejecución de un programa. Algunas de ellas son específicas de cada hebra, mientras que otras son comunes a todas. Por tanto, algunas se construyen al principio de la ejecución de la máquina virtual, y otras se crean cuando la aplicación crea hebras:

- **Heap**: es una zona global, común a todas las hebras, donde se guardan las instancias de las clases y los vectores (*arrays*) creados. En la especificación no se establece ninguna restricción a cómo manejar este espacio de memoria (si es de tamaño fijo o no, etc.), así como no se hace mención a cómo almacenar cada una de las instancias de las clases, aunque sí indica que debe ser controlado por un recolector de basura que elimine los objetos que no son accesibles desde la aplicación (accesibles desde ningún método en ejecución).
- **Área de métodos**: es donde se guarda el código de cada uno de los métodos de las clases cargadas en la máquina virtual.
- **Pilas de la JVM**: cada una de las hebras tiene una pila. En ella se guardan los registros de activación de los métodos pendientes de ejecutar que, en nomenclatura de la JVM, se llaman *frames*. Por su importancia, los detallaremos más adelante.
- **Contador de programa**: también es privado para cada hebra, e indica qué instrucción se está ejecutando. En cada método se reinicia la cuenta, es decir, la primera instrucción de un método es siempre la instrucción cero. El contador de programa indica qué instrucción se debe ejecutar dentro del método actual. La información sobre cuál es el método no se guarda en el contador de programa, sino en cada *frame*.
- **Área de constantes**: en cada fichero `class` existe un área de constantes con nombres y valores que son referenciados en los campos de dichas clases o desde las instrucciones de cada método. De esta forma, una instrucción de llamada a un método, hace referencia a una entrada en esa tabla de constantes que contendrá el nombre del método, en vez de tener el propio nombre como argumento. Así, los nombres son compartidos por distintas instrucciones, ahorrando espacio en disco.

- **Pila para métodos nativos**: la máquina virtual permite hacer llamadas a métodos no implementados en Java, cargados de una librería externa. Estos métodos son conocidos como métodos *nativos*, ya que han sido compilados para la máquina *real* en la que se ejecuta la JVM. Para su ejecución, se dispone de una pila para cada hebra.

Los registros de activación o *frames* son el elemento más importante para la ejecución de los métodos de Java. Un *frame* permite almacenar datos, resultados parciales, valores devueltos y lanzar excepciones. Cada vez que se llama a un método, un nuevo *frame* es creado en la pila de la hebra. Ese *frame* posee un área para almacenar las variables locales del método, una pila de operandos y una referencia a la tabla de constantes del método que se está ejecutando. El tamaño de las variables locales y de la pila de operandos se conoce de antemano, y es guardado por el compilador en el fichero compilado (y más adelante comprobado en tiempo de carga), por tanto, el *frame* puede utilizar estos datos para no desaprovechar espacio.

Una vez más, debemos hacer notar que estas estructuras son las que marca la especificación. En este sentido, el documento divide la máquina virtual en estas partes y explica el funcionamiento de la JVM a partir del funcionamiento de cada una de ellas, indicando con qué valores son inicializadas, cuándo se escribe en ellas y cuándo se utilizan. Sin embargo, la implementación particular de una JVM puede prescindir de alguna de las estructuras debido a que la funcionalidad que dan la consiguen de otra forma. Por ejemplo, en muchas implementaciones el área de constantes es suprimida; los campos de las instrucciones, en vez de tener una referencia a ese área, contienen directamente la constante, para un acceso más rápido. Si el funcionamiento de cara al exterior es el indicado por la especificación, se puede decir que la implementación cumple esa especificación, independientemente de si internamente tiene las estructuras marcadas o no.

A.2.3. Conjunto de instrucciones

Uno de los aspectos que más llama la atención de las instrucciones máquina de la JVM es el gran número de instrucciones dedicadas para cada una de las operaciones. Así, existen múltiples instrucciones de suma, resta, etc. La razón es simple: cada una de ellas permite operar con valores de distintos tipos, por lo que hay una instrucción para sumar enteros, otra para sumar enteros largos, otra para reales, reales de doble precisión, etc.

Cuando la operación admite distintos tipos, para distinguir entre todas las versiones, el mnemónico dispone de un *prefijo* que indica el tipo, según la nomenclatura de la tabla A.1.

Las instrucciones pueden agruparse en varias categorías que explicamos a continuación.

Prefijo	i	l	s	b	c	f	d	a
Tipo	int	long	short	byte	char	float	double	reference

Tabla A.1: Prefijos utilizados en las instrucciones de la JVM

A.2.3.1. Operaciones de carga y almacenamiento

Son las responsables de mover valores desde las variables locales a la pila de operandos (carga) o viceversa (almacenamiento).

La carga se realiza con las instrucciones *{i,l,f,d,a}load* o *{i,l,f,d,a}load_[n]*. En el primer caso, el número de la variable local a cargar es un parámetro de la instrucción, y en el segundo caso viene en el código de la instrucción (n entre 0 y 3).

El almacenamiento es equivalente, utilizándose en este caso las instrucciones *{i,l,f,d,a}store* o *{i,l,f,d,a}store_[n]*.

Existen otras operaciones que cargan valores constantes en la pila. Esos valores pueden ser operandos de la instrucción, o estar codificados en el opcode. Por ejemplo existen instrucciones específicas para apilar valores enteros entre -1 y 5 (*iconst_m1*, *iconst_[i]*), para apilar el 0 y el 1 de tipo `long`, etc.

A.2.3.2. Instrucciones aritméticas

Como norma general, cogen dos operandos de la pila de operandos, operan con ellos y dejan en la cima el resultado. No existen operaciones para los tipos `byte`, `short`, `char` ni `boolean`. Para operar con ellos, se utilizan las instrucciones que trabajan con los enteros.

Las operaciones son *{i,l,f,d}add*, *{i,l,f,d}sub*, *{i,l,f,d}mult*, *{i,l,f,d}div*, *{i,l,f,d}rem* (módulo), *{i,l,f,d}neg*, *{i,l}or*, *{i,l}and*, *{i,l}xor* e *iinc* (incrementa una variable local).

Además, hay operaciones de desplazamiento a nivel de bits de enteros y de enteros largos, tanto a derecha como a izquierda y teniendo en cuenta el signo o no: *{i,l}shl*, *{i,l}shr*, *{i,l}shr* (no existe *{i,l}ushl*).

Por último, existen instrucciones de comparación que dejan en la pila un valor entre -1 y 1 dependiendo del resultado.

A.2.3.3. Instrucciones de conversión de tipos

Cogen de la pila el dato, lo convierten al tipo destino y lo vuelven a apilar. Las instrucciones se representan de la forma *a2b*, donde *a* significa el tipo origen, y *b* el destino (y 2 representa la preposición inglesa "*to*").

Las instrucciones para conversión de tipos que soporta la máquina virtual son *i2l*, *i2f*, *i2d*, *l2f*, *l2d*, *f2d*, *i2b*, *i2c*, *i2s*, *l2i*, *f2s*, *f2l*, *d2i*, *d2l* y *d2f*.

A.2.3.4. Creación y acceso a objetos

Existen instrucciones específicas para la creación de objetos y de *arrays*, así como para el acceso a cada posición de estos últimos:

- Creación de una instancia de una clase: *new*.
- Creación de un *array* (vector) de tipos primitivos, de referencias o un array multidimentsional: *newarray*, *anewarray*, *multianewarray*.
- Acceso a los campos de las clases (estáticos) y de los objetos: *getfield*, *putfield*, *getstatic*, *putstatic*.
- Acceso a un componente de un *array*, y almacenarlo en la pila de operandos: *{b,c,s,i,l,f,d,a}aload*.
- Almacenar un componente de un *array* desde la pila de operandos: *{b,c,s,i,l,f,d,a}astore*.
- Obtener la longitud de un vector: *arraylength*.
- Comprobar si una referencia apunta a una clase o tipo de *array* determinado: *instanceof*, *checkcast*.

A.2.3.5. Instrucciones con la cima de la pila

Algunas instrucciones operan directamente sobre la cima de la pila. Por ejemplo, *pop* y *pop2* eliminan uno o dos elementos de la cima de la pila, mientras que *swap* los intercambia.

Por último, existe una batería de instrucciones que permiten duplicar algunos elementos de la cima de la pila. Dependiendo del número de ellos (uno o dos) y de dónde lo inserte (en la cima de la pila, o inmediatamente debajo de ella), disponemos de *dup*, *dup2*, *dup_x1*, *dup2_x1*, *dup_x2* y *dup2_x2*.

A.2.3.6. Instrucciones de salto

En este grupo situamos tanto los saltos condicionales como los incondicionales. La JVM también dispone de unas instrucciones específicas creadas para la compilación de la instrucción `switch`.

- Saltos condicionales: existe un grupo de saltos condicionales que saltan o no dependiendo del resultado de la comparación de un entero con el valor cero (*ifeq*, *iflt*, *ifle*, *ifne*, *ifgt* y *ifge*), o del resultado de la comparación de dos enteros en la pila (*if_icmpeq*, *if_icmpne*, *if_icmplt*, *if_icmpgt*, *if_icmple*, *if_icmpge*, *if_cmpeq* y *if_cmpne*). Por último, también existen dos saltos que dependen de si una referencia es `null` o no (*ifnull* y *ifnonull*).

- Salto incondicional: además del común *goto* (y su versión con dirección ancha, *goto_w*), existen tres instrucciones para los saltos incondicionales necesarios debido a la construcción `finally` de Java, *jsr*, *jsr_w* y *ret*.

- Instrucción `switch`: dependiendo de lo dispersos que sean los distintos casos o posibilidades dentro de los `case`, se utiliza *tableswitch* o *lookupswitch*.

Las instrucciones de invocación a métodos también pueden considerarse instrucciones de salto, ya que rompen la ejecución en orden del programa. Sin embargo, dada su importancia, las tratamos de modo independiente.

A.2.3.7. Invocación de métodos

Existen cuatro instrucciones distintas para hacer llamadas métodos. Las diferencias entre ellas (especialmente entre las tres primeras) son bastante *sutiles*, ya éstas radican principalmente en cómo buscan el método en la jerarquía de clases.

- *invokevirtual*: Llama al método de un objeto, haciendo la resolución del método que se espera en un lenguaje orientado a objetos, es decir, si el método no existe en la clase a la que pertenece el objeto, lo busca en la clase padre.

- *invokeinterface*: Se utiliza para llamar a un interfaz. Funciona de forma parecida al *invokevirtual*, buscando la implementación en la clase a la que pertenece el objeto que recibe el mensaje y, si no existe, en la clase padre. La diferencia fundamental son las comprobaciones de acceso realizadas. En este caso, la instrucción recibe no solo el nombre del método, sino también el nombre del interfaz al que pertenece, comprueba que la clase implementa ese interfaz y alguna otra comprobación inherente a este tipo de invocaciones.

- *invokespecial*: como su nombre indica sirve para realizar llamadas *especiales* a distintos métodos. Por ejemplo, se utiliza para llamar a los constructores de las clases, a métodos privados o a un método de la clase padre a la que pertenece el objeto.

- *invokestatic*: utilizado para ejecutar métodos estáticos de una clase.

A.2.3.8. Instrucciones para excepciones

La única instrucción relacionada con el lanzamiento de excepciones es *athrow*, que recibe la referencia al objeto de la excepción que se lanzará.

La captura de excepciones, al ser un mecanismo controlado por la JVM y externo a la propia ejecución de las instrucciones, se encuentra situado en la información estática del propio método.

Para la implementación del `finally`, ya hemos comentado que se utilizan las instrucciones de salto incondicional *jsr*, *jsr_w* y *ret*.

A.2.3.9. Instrucciones de sincronización

Java permite al programador la ejecución de varias hebras concurrentemente, y la existencia de monitores para su programación segura.

Cada objeto Java posee un *monitor* utilizado para la sincronización. Para el control de estos monitores, la máquina virtual dispone de dos instrucciones para bloquearlos y desbloquearlos, que son *monitorenter* y *monitorexit* respectivamente.

Apéndice B

Gestor de scripts en Java desde C++

Este apéndice contiene una pequeña descripción del módulo `ScriptSystem` de JV^{2}M, que permite la invocación de métodos creados en clases Java desde una aplicación como la nuestra, implementada en C++.

B.1. Introducción

Aunque JV^{2}M está implementado en C++, existe una parte de la implementación del subsistema de tutoría (sección 6.6) que está implementada en Java. En particular, decíamos en la página 203 que el sistema de ayuda de JV^{2}M 2 utiliza jCOLIBRI para la implementación de la ayuda semántica y sintáctica.

Para la comunicación entre C++ y Java, JV^{2}M dispone de un módulo llamado `ScriptSystem` que está asociado al módulo de tutoría, y que se inicializa cuando éste. El módulo consta de una serie de clases que crean una instancia de la máquina virtual, permiten la carga de clases Java, la creación de objetos y la invocación a sus métodos.

Los siguientes apartados describen lo más significativo del módulo.

B.2. JNI: Java Native Interface

Los diseñadores de Java crearon un mecanismo para conseguir que las aplicaciones de Java pudieran llamar a métodos de C/C++. Este mecanismo

utiliza una colección de APIs de C++ conocidos como Java Native Interface (JNI). Así, se puede llamar a métodos denominados nativos (implementados en C/C++), desde Java.

Gran cantidad de los métodos de la librería de Java están implementados haciendo uso de estos métodos nativos. Por ejemplo, la lectura de ficheros, soporte de red, dibujado en pantalla, etc. Todas estas funcionalidades de Java están escritas en C/C++, haciendo uso del sistema operativo subyacente.

Por lo tanto, la realidad es que JNI fue una "estandarización" de un interfaz que los propios desarrolladores de Sun necesitaban para poder implementar toda la librería de Java[1].

Además, los métodos desarrollados en C/C++ pueden a su vez, basar su implementación en llamadas a otros métodos Java. Es decir, JNI permite desde C++ llamar a métodos Java. Esta capacidad es la que se aprovecha en JV^{2}M, dando la vuelta a la utilidad inicial de JNI: en vez de utilizarlo para, desde aplicaciones Java, llamar a métodos en C++, lo usaremos para poder llamar a Java desde aplicaciones en C++.

JNI es distribuido con la máquina virtual de Java desde la versión 1.1 del JDK. Viene en forma de librería dinámica (DLL o .so), que tiene funciones que pueden invocarse desde una aplicación en C/C++. De hecho, la forma normal de ejecutar la máquina virtual (ejecutable `java`), no es más que un simple programa de C que interpreta los argumentos de la línea de comandos, y llama a la máquina virtual de Java utilizando esa DLL que implementa el interfaz JNI.

Al instalar JDK 1.5, se instala (en el directorio `$JDKROOT$/include`) el .h que contiene las funciones de JNI. Además, el proyecto en C++ debe incluir en el enlazado el fichero `jvm.lib` (que aparece en el directorio `$JDKROOT$/lib`), para que enlace estáticamente la DLL con la implementación de la JVM.

El programa debe inicialmente crear una máquina virtual utilizando una función del JNI. Después cargar las clases que necesite, y ejecuta los métodos deseados. Por último, destruye la máquina virtual, para eliminarla de la memoria.

El proceso de creación de una JVM es relativamente costoso en tiempo, por lo que JV^{2}M la arranca al principio de su ejecución, y la mantiene activa hasta su finalización.

Para ocultar las peculiaridades de JNI al resto del sistema, hemos implementado la clase `JavaIntfz::JVM`, que utiliza JNI para crear y destruir la máquina virtual. Además, sobrecarga el operador flecha de C++ (`->`), para poder llamar a las funciones de JNI utilizando un objeto de la clase. El código de la clase aparece en la sección C.2.1.

[1] Y de hecho, fue uno de los puntos de conflicto con Microsoft, que intentaron establecer su propio mecanismo de comunicación, para facilitar las llamadas a objetos COM.

B.2.1. Soporte para hebras

La máquina virtual de Java permite que distintas hebras se ejecuten sobre ella. Es más, también permite que varias hebras de C++ invoquen a métodos suyos simultáneamente.

Lo normal es utilizar JNI de tal forma que la máquina virtual tenga el control completo de la aplicación, y de vez en cuando llame a métodos nativos de C++. Eso puede provocar que, si en Java se han creado hebras, se estén ejecutando *simultáneamente* varios métodos nativos.

En nuestro caso, la aproximación es a la inversa, por lo que el tratamiento de hebras también. Más concretamente, estamos utilizando Java desde C++, es decir, en C++ hemos creado una JVM, y llamamos a métodos de sus objetos. Esa aplicación (JV^2M) puede crear distintas hebras en C++, y llamar a métodos Java desde todas ellas.

Para hacerlo posible, la JVM, y en particular JNI, requiere que cada nueva hebra que quiera llamar a la JVM sea previamente reconocida por la JVM, o mejor dicho sea "vinculada" (*attach*) a la JVM, llamando a un método de JNI denominado `AttachCurrentThread`. Si una hebra no vinculada a la JVM llama a alguno de los métodos, provocará la interrupción abrupta del programa.

El soporte de hebras en JNI se realiza mediante el "*entorno de ejecución*" (`JNIEnv`). En la creación de la JVM, se crea el primer entorno automáticamente para la hebra invocante. Si se quieren otras hebras, hay que indicárselo a la JVM (mediante la corespondiente llamada a JNI), para que se cree otro entorno de ejecución (`JNIEnv`) para esa nueva hebra.

Cuando una hebra no va a volver a utilizar la máquina virtual, tiene que *desvincularse* de ella, utilizando el método `DettachCurrentThread`, que elimina o libera ese entorno de ejecución.

La clase `JVMIntfz::JVM` proporciona soporte para hebras, permitiendo a distintas hebras (que en JV^2M creamos con la librería `pthread`) vincularse y desvincularse de la máquina virtual. El operador `->` anteriormente citado detecta en qué hebra se está, y devuelve el entorno de ejecución de esa hebra, de tal forma que las llamadas a JNI se realizan automáticamente utilizando el entorno adecuado.

B.3. ScriptManager

El gestor de los scripts está implementado en la clase `CScriptManager`. Es un `singleton` con inicialización y destrucción explícita de la única instancia, mediante los métodos estáticos `Init` y `Release`, para evitar problemas por el orden de inicialización de los singleton.

El objeto gestor en su inicialización crea una máquina virtual de Java, poniendo como classpath el directorio `./scripts`, donde se encuentran las

clases Java que utiliza JV^{2}M.

El gestor permite la carga de clases en la máquina virtual (ver el apartado B.3.1), y la construcción de objetos a partir del nombre de la clase (apartado B.3.2), así como la invocación tanto de métodos estáticos como dinámicos (ver la sección B.5). Por último, dada la importancia de las cadenas en el lenguaje Java (`java.lang.String`), también dispone de una clase para la traducción sencilla entre ellas y las cadenas nativas de C++ (`std::string`).

El código de la definición de la clase aparece en la sección C.2.2.

B.3.1. ScriptClass

La clase `CScriptClass` (ver su definición en el apartado C.2.3) representa una clase de la máquina virtual de Java. Cuando el gestor de la máquina virtual, `CScriptManager`, carga una clase en Java, devuelve como "manejador" un puntero a un objeto de `CScriptClass`.

En realidad, el usuario del sistema de scripts *no* puede crear un objeto de esta clase, sino que siempre tiene que hacerlo a través del `CScriptManager`. El método `CScriptManager::getScriptClass` devuelve un puntero a un objeto `CScriptClass` que *no es propiedad del invocador*, es decir, el usuario *nunca* tendrá que liberar este tipo de objetos[2].

La clase tiene dos funciones principales:

- La creación de objetos de la propia clase: llamando al constructor por defecto, y devolviendo un objeto de la clase C++ `CScriptObject` que describiremos en el apartado B.3.2.

- Invocación a métodos estáticos. Las llamadas a los métodos (en general, tanto estáticos como de instancia) tiene la importancia suficiente como para que le dediquemos el apartado B.5 completo, por lo que no lo describimos aquí.

Desde el punto de vista de la implementación interna del gestor de scripts, la clase `CScriptClass` también es la responsable de buscar si existen los métodos pedidos (a partir del nombre del método y su signatura) en la clase, y de guardar las referencias JNI en una caché interna.

B.3.2. ScriptObject

La clase `CScriptObject` (ver su definición en la sección C.2.4) simboliza un objeto de la máquina virtual de Java. Internamente, el `CScriptObject` almacena una referencia JNI al objeto de la máquina virtual (`jobject`), que

[2]Como hemos dicho anteriormente, el `CScriptManager` mantiene internamente una lista de todas las clases cargadas, y se liberan al final.

libera en su destructor. Eso provoca que si la JVM lo considera oportuno, sea liberado por el recolector de basura.

Según esto, dos objetos de `CScriptObject` podrían referenciar al mismo objeto Java (ya veremos un ejemplo de cómo puede darse esto en el apartado B.5 de invocación de métodos). La sobrecarga del operador `==` se asegura de que la comparación entre objetos C++ se haga de forma correcta (comprobando si representan al mismo objeto Java).

Para crear un objeto Java y conseguir un puntero a un objeto C++ de esta clase, el usuario del módulo utilizará el método `CrearObjeto` de la clase `CScriptManager` o de `CScriptClass` (el primero recibe el nombre de la clase). Ambos piden memoria para el objeto y devuelven un puntero a un objeto de esta clase.

Los métodos disponibles para invocar a los métodos de Java son descritos en el apartado B.5.

B.4. Cadenas en Java

Las cadenas son un tipo casi primitivo en Java. A pesar de que son una clase propia dentro del classpath (`java.lang.String`), su mera localización en el paquete `java.lang` parece indicar que es una clase con privilegios dentro de la JVM.

Por esa razón, y porque es una clase muy utilizada, el módulo de comunicación con Java dispone de una clase, `CStringScript`, que facilita su uso. Esta clase hereda de `CScriptObject`, por lo tanto se puede llamar a todos sus métodos de la misma forma que con cualquier otro objeto. La ventaja de utilizar esta clase especializada es que dispone de un nuevo método, `getCadena`, que devuelve la cadena en C++ (`std::string`). El propio gestor, además, tiene un método para crear una cadena en vez de un objeto genérico, a partir de su representación en C++.

B.5. Invocación de métodos

La parte más importante del interfaz con Java es la posibilidad de invocar métodos. Aunque en este apartado hablaremos de los métodos de objeto (o de instancia), las explicaciones se pueden extrapolar a los métodos estáticos (la diferencia es que desde C++, unos se llaman utilizando la clase `CScriptObject` y los otros `CScriptClass`).

Para llamar a los métodos Java, se utiliza la familia de métodos `InvokeXXXX` (o `InvokeStaticXXX`). La lista completa es la siguiente:

```
void InvokeVoid(const char *name,
                const char *signature, ...);
bool InvokeBoolean(const char *name,
```

```
                const char *signature, ...);
int InvokeInt(const char *name, const char *signature, ...);
double InvokeDouble(const char *name,
                    const char *signature, ...);
float InvokeFloat(const char *name,
                  const char *signature, ...);
CStringScript* InvokeString(const char *name,
                            const char *signature, ...);
CScriptObject* InvokeObject(const char *name,
                            const char *signature, ...);
```

Todos ellos reciben como primer parámetro el nombre del método Java, como segundo parámetro la signatura (o prototipo), y a continuación una lista de parámetros variable (y opcional), que serán pasados al método Java invocado.

Para indicar el tipo (en el segundo parámetro) se utiliza una cadena con una codificación especial, que permite representar cualquier tipo de Java (ya sean tipos básicos, clases o métodos). La tabla B.1 refleja el método de codificación y el ejemplo de un método con tres parámetros (un entero, una cadena y un *array* de enteros) que devuelve un valor binario.

Hay que hacer notar que, obviamente, los parámetros que se indican en la función de C++ son valores *en C++*. Por lo tanto, si se llama a un método que recibe un booleano, se pasará como parámetro una variable o valor booleana en C++, que tendrá su correspondiente tamaño en bytes. Lo mismo si se tienen que pasar `int`s, `float`s, etc. Esta característica puede ser un foco de problemas cuando el tamaño de los datos en C++ no es el mismo que el de Java (como ocurre con los tipos `long` o `char`). Debido a esto, el gestor de comunicación *no* soporta aquellos tipos de datos que tienen rango de representación distinto en ambos mundos.

Si un método recibe como parámetro una referencia a un objeto, se utiliza como parámetro el *puntero al `CScriptObject`*. La implementación interna del método `InvokeXXXX`, hace la conversión de C++ a Java antes de llamar a JNI.

Los métodos Java pueden devolver distintos tipos (o nada, o `bool`, etc.). Por eso, hay un método por cada posible tipo devuelto. Destaca también un método para llamar a métodos Java que devuelvan `String`, para luego poderla manejar con `CStringScript`, en vez de con el genérico `CScriptObject`.

Por último, hay que decir que el `InvokeObject` es lo suficientemente inteligente como para construir un nuevo `CScriptObject` con el valor devuelto. Eso hace posible que dos `CScriptObject` distintos referencien al mismo objeto Java. Por ejemplo, se consigue con la simple llamada a un método (poco útil pero válido como ejemplo) como "`myself`".

```
ClasePrueba myself() { return this; }
```

Signatura	Tipo de Java
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
Lpaquete/paquete/nombre_clase;	clase
[signatura	tipo[]
(signaturas)tipo_devuelto	tipo_devuelto (signaturas) (función)
Ejemplo	
`(ILjava/lang/String;[I)Z`	`boolean` *`func`* `(int, String, int [])`

Tabla B.1: Codificación de tipos de Java en cadenas.

B.6. Uso de Java en JV^{2}M

Como hemos comentado anteriormente, el subsistema de tutoría de JV^{2}M utiliza Java para implementar la ayuda semántica que se ofrece al usuario. La ayuda, implementada en el subsistema de tutoría, se inicializa al principio de la aplicación. Desde C++ está representada por la clase `CHelp`, que contiene los siguientes métodos:

- `InitHelp`: comienza la inicialización de la ayuda. Provoca la creación de la clase Java `help.Help`, que, utilizando jCOLIBRI, implementa la ayuda sintáctica y semántica. Ésta inicialización es bastante costosa en tiempo, por lo que se ejecuta en una hebra distinta que aprovecha el soporte de hebras visto en el apartado B.2.1.

- `ReleaseHelp`: elimina el objeto Java, para que la máquina virtual pueda liberar todos sus recursos.

- `InitializationComplete`: devuelve un valor binario que indica si la ayuda se ha terminado de inicializar (es decir, la hebra secundaria ha terminado).

- `AskForHelp`: recibe una cadena en C++ (`std::string`), que traduce a una cadena Java para invocar al método con el mismo nombre de la clase java `help.Help`. Éste devuelve a su vez la cadena (Java) con el mensaje a proporcionar al estudiante que es a su vez convertido a una cadena C++ que se presenta al usuario.

Apéndice C

Código representativo de JV^{2}M

Este apéndice contiene secciones representativas de código C++ de JV^{2}M, referenciadas desde el texto.

C.1. Motor de la aplicación

C.1.1. `Maps::CMapFile`

```
/**
 @file MapFile.h

 Contiene la declaración de la clase que representa una mapa
 cargado desde un fichero.

 La clase contiene toda la información leída del mapa, de tal
 forma que puede posteriormente utilizarse para lanzar el mapa
 (creando los objetos que gestionan todas las entidades).

 @author Marco Antonio Gómez Martín
*/

#ifndef MapFile_H
#define MapFile_H

#include <vector>
#include <string>

// Predeclaración de clases
namespace Maps {
   class CMapEntity;
```

```
    class CMapModel;
    class CMapLoader;
}

namespace Maps {

/**
 Clase que almacena la información completa de un mapa leído de
 disco.

 El constructor de la clase es protegido, por lo que la única
 forma que existe de crear objetos es utilizando el método
 factoría createMapFromFile. Este método puede analizar la
 extensión del fichero, y delegar la carga del fichero a otras
 clases.

 La clase almacena una lista con todas las entidades del fichero,
 así como una lista con todos los modelos que tiene.

 @author Marco Antonio Gómez Martín
*/
class CMapFile {
public:

    // Clase CMapLoader friend, para poder tener acceso a la
    // lista de entidades.
    friend class CMapLoader;

    /**
     Carga un mapa desde fichero, y devuelve la clase que lo
     representa. Si no se ha podido cargar el mapa, devuelve NULL.
     El método no comprueba si ese mapa ya se cargó en algún
     momento anterior, por lo que debe existir un gestor externo
     que mantenga los mapas ya cargados, para evitar
     duplicaciones.
     @param name Nombre del mapa que se carga.
     @return Puntero al objeto que contiene la información del
     mapa.
     @note Para la carga, utiliza la extensión del fichero de tal
     forma que dependiendo de la misma, redirige la interpretación
     del fichero a una u otra clase.
    */
    static CMapFile *createMapFromFile(const std::string &name);

    /**
     Métodos que devuelve el nombre del mapa.
     @return nombre del mapa.
    */
    const std::string &getName() const { return _nombreMapa; }

    /// Destructor de la clase. Es virtual puro, para hacer la
    /// clase abstracta.
    virtual ~CMapFile() = 0;
```

```
protected:

    /// Constructor de la clase. Es protegido; la única forma de
    /// crear objetos es invocando a createMapFromFile.
    CMapFile() {}

    /// Tipo de datos para guardar la colección de entidades.
    typedef std::vector<Maps::CMapEntity*> TListaEntidades;

    /// Colección de entidades del mapa
    TListaEntidades _listaEntidades;

    /// Tipo de datos para guardar la colección de modelos.
    typedef std::vector<Maps::CMapModel*> TListaModelos;

    /// Colección de los modelos del mapa
    TListaModelos _listaModelos;

    /// Nombre del mapa
    std::string _nombreMapa;
};

} // namespace Maps

#endif // MapFile_H
```

C.1.2. Maps::CMapEntity

```
/**
 @file MapEntity.h

 Contiene la declaración de la clase que representa una entidad
 cargada de un mapa.

 @author Marco Antonio Gómez Martín
*/

#ifndef MapsMapEntity_H
#define MapsMapEntity_H

#include <map>
#include <string>

#include "Base/Vector3.h"

// Predeclaración de clases que serán amigas de esta:
// todos los cargadores de mapas. Así, podrán crear objetos de
// este tipo. También el modelo de un mapa, ya que las entidades
// guardan punteros a sus modelos (si los tienen).
namespace Maps {
    class CMapBSPFile;
    class CMapModel;
    class CMapFile;
}
```

```
namespace Maps {

/**
Clase que almacena la información de una entidad de un mapa.

Una entidad de un mapa es simplemente una lista de atributos
clave-valor. El valor puede tener distinto tipo, por lo que la
clase tiene métodos para acceder a valores de distinto tipo.

Los objetos de esta clase son creados en el momento de cargar el
mapa concreto desde fichero, y luego no debería modificarse. Por
eso todos los métodos de la clase que alteran su contenido son
protegidos, y únicamente pueden acceder a ellos las clases
amigas.

La forma de guardar las parejas atributo-valor es bastante
primitiva: se guardan en una tabla hash cuya clave es el nombre
del atributo, y cuyo valor es un simple buffer (char*) que,
dependiendo de lo que quiera el usuario se convierte al vuelo.

Las entidades NO son las encargadas de guardar los modelos (si
los hay) almacenados en el mapa. De eso se encarga otra clase
distinta (CMapModel). Las entidades, en caso de tener modelos,
almacenan un puntero a un objeto de esa clase que, en ningún
caso, es propiedad suyo (es decir, no es borrado por la clase en
el destructor).
*/
class CMapEntity {
public:

   /**
    Sirve para comprobar si un atributo existe o no.
    @param atrib Nombre de la clave/atributo.
    @return Devuelve true si el atributo existe.
    */
   bool existsKey(const std::string &atrib) const;

   // Métodos de acceso generales a los atributos

   /**
    Devuelve el valor de un atributo. Normalmente el
    atributo será una cadena terminada con <code>'\\0'</code>,
    que el cliente de la clase podrá utilizar.
    @param nombre Nombre del atributo por el que se pregunta.
    @return Puntero constante al valor del atributo. El método
    que invoca este método NO tiene la propiedad de ese puntero;
    puede utilizar su valor mientras dure la vida del propio
    atributo, pero no después. Tampoco debe  modificar el valor,
    ni eliminarlo.
    @note Si el atributo no existe, devuelve NULL.
    */
   const char *getAtributo(const std::string &nombre) const;
```

```
/**
 Devuelve el valor de un atributo de tipo real.
 @param nombre Nombre del atributo.
 @return Valor real del atributo.
 @note Si el atributo no existe, se devuelve 0.0f
 */
float getAtributoFloat(const std::string &nombre) const;

/**
 Devuelve el valor de un atributo de tipo entero.
 @param nombre Nombre del atributo.
 @return Valor entero del atributo.
 @note Si el atributo no existe, se devuelve 0.
 */
int getAtributoInt(const std::string &nombre) const;

/**
 Devuelve el valor de un atributo de tipo vector.
 @param nombre Nombre del atributo.
 @return Valor del atributo.
 @note Si el atributo no puede interpretarse como vector,
 o no existe, se devuelve (0.0f, 0.0f, 0.0f)
 */
Base::TVector3 getAtributoVector(const std::string &nombre)
  const;

// Métodos de acceso a atributos "especiales"

/**
 Devuelve el tipo de la entidad o clase de entidad.
 @return Referencia constante a la cadena que contiene el
 nombre de la clase de la entidad.
 */
const std::string &getEntityType() const;

/**
 Devuelve la posición de la entidad si esta es puntual. Si no
 lo es, devuelve (0, 0, 0).
 @return Posicion de la entidad, si es que existe.
 */
Base::TVector3 getOriginalPosition() const;

/**
 Permite acceder al modelo de la entidad contenido en el
 fichero del mapa origen (si es que lo tiene).
 @return Puntero al objeto que representa el modelo de la
 entidad (NULL si no existe).
 */
const CMapModel *getModelo() const;

/// Destructor de la clase
~CMapEntity();
```

```
protected:

   /// Constructor de la clase protegido, para que sólo el
   /// cargador del mapa pueda crearlo
   /// @param mapa El objeto que representa el mapa, por si
   /// alguna subclase lo necesita para poder implementar alguno
   /// de los métodos.
   CMapEntity(const CMapFile *mapa);

   /**
    Establece el valor de una clave. Hace una copia del valor que
    recibe (segundo parámetro), que interpreta que termina con
    el '\\0'.
    @param atributo Nombre del atributo/clave que se establece.
    @param valor Valor que se quiere utilizar.
   */
   void setKeyValue(const std::string &atributo,
          const char *valor);

   // Funciones para establecer los valores de los atributos
   // "especiales"

   /**
    Establece el tipo de entidad.
    @param tipo Tipo de la entidad.
    */
   void setEntityType(const std::string tipo);

   /**
    Establece la posición origen de la entidad.
    @param pos Posición inicial de la entidad.
    */
   void setOriginalPosition(const Base::TVector3 &pos);

   /**
    Establece el modelo inicial de la entidad.
    @param modelo Modelo inicial.
    */
   void setModelo(const CMapModel *modelo);

   // Clases amigas: todos los cargadores de mapas
   friend class CMapBSPFile;

private:

   /// Tipo de datos privado para guardar la tabla con la
   /// información
   typedef std::map<std::string, const char *> TTablaAtributos;

   /// Tabla que contiene los atributos de la entidad.
   TTablaAtributos _atribs;

   // Atributos "especiales" de la entidad, que no se guardan
   // en la tabla.
```

```
    /// Tipo de entidad
    std::string _entityType;

    /// Posición inicial de la entidad
    Base::TVector3 _origenInicial;

    /// Puntero al modelo.
    const CMapModel *_modelo;

    /// Constructor copia no implementado
    CMapEntity(const CMapEntity &);
};

} // namespace Maps

#endif // MapsMapEntity_H
```

C.1.3. Maps::CMapModel

```
/**
 @file MapModel.h

 Contiene la declaración de la clase que representa un modelo
 cargado desde un fichero que contiene un mapa.

 La clase no permite en principio acceder físicamente al modelo,
 sino que tiene métodos para poder crear otras clases que lo
 utilicen (entidad gráfica y de colisión).

 @author Marco Antonio Gómez Martín
 @date Marzo, 2007
*/

#ifndef MapsMapModel_H
#define MapsMapModel_H

// Predeclaración de clases

namespace Graphics {
    class CEntity;
}

namespace Collision {
    class CCollisionShape;
    class CCollisionContext;
}

// Declaración de la clase

namespace Maps {

/**
Clase que almacena la información de un modelo contenido en un
```

```
fichero de mapa.

Normalmente estos modelos representan geometría estática del mapa
(como las paredes), y su representación geométrica puede estar
optimizada utilizando alguna técnica específica (típicamente,
BSPs).

Los objetos de esta clase son creados en el momento de cargar el
mapa concreto desde fichero, y luego no debería modificarse.

Los métodos que tiene la clase no permiten en realidad acceder a
todas sus propiedades, sino que únicamente permiten crear clases
que representan a ese modelo en los distintos módulos del juego.
Así, es posible crear un objeto que lo represente gráficamente, y
otro que lo represente en el mundo de colisión.

Existe una clase hija de ésta por cada tipo de mapa que se
pueda cargar.
*/
class CMapModel {
public:

   /**
    Crea la entidad gráfica asociada al modelo.
    @return Entidad gráfica asociada al modelo. Dependiendo de
    sus características, la entidad gráfica será un BSP, una
    maya estática, etc. Devuelve NULL si hay algún problema en
    su creación.
    */
   virtual Graphics::CEntity *createGraphicsEntity() const {
     return 0;
   }

   /**
    Crea una forma de colisión asociada al modelo.
    @return Forma de de colisión asociada al modelo, o NULL si
    hay algún error.
    @note La forma de colisión deberá ser añadida a algún mapa
    de colisión en forma de objeto.
    */
   virtual Collision::CCollisionShape *createCollisionEntity()
     const { return 0; }

   /// Destructor de la clase
   virtual ~CMapModel() {};

protected:

   /// Constructor de la clase protegido, para que sólo el
   /// cargador  del mapa pueda crearlo
   CMapModel() {}

private:
   /// Constructor copia no implementado
```

```
    CMapModel(const CMapModel &);
};

} // namespace Maps

#endif // MapsMapModel_H
```

C.1.4. Maps::CMapLoaderObserver

```
/**
 @file MapsLoaderObserver.h

 Definición de la clase observer del cargador de mapas
 Maps::CMapLoader.

 @author Marco Antonio Gómez Martín
*/
#ifndef MapsLoaderObserver_H
#define MapsLoaderObserver_H

#include <string>

// Predeclaración de clases
namespace Maps {
    class CMapEntity;
}

// Definición de la clase

namespace Maps {

/**
 Interface que implementa el código específico para ser avisado
 por el cargador de mapas cuando se procede a la carga de uno de
 ellos.
 @author Marco Antonio Gómez Martín.
*/
class CMapLoaderObserver {
public:
    /**
    Función llamada cuando se va a empezar la carga del mapa.
    @param nombreMapa Nombre del mapa
    */
    virtual void OnBeginMapLoading(const std::string &nombreMapa)
      = 0;

    /**
    Función llamada cuando se ha terminado la carga del mapa.
    */
    virtual void OnEndMapLoading() = 0;

    /**
    Función llamada por el cargador por cada una de las entidades
    del mapa que se está cargando.
```

```
    @param entityInfo Información sobre la entidad.
    @return Devuelve false si no se ha podido realizar la creación
    del mapa, y por lo tanto, se debe cancelar la carga completa.
    */
    virtual bool InitEntity(const Maps::CMapEntity *entityInfo)
      = 0;
};

} // namespace Maps
#endif
```

C.1.5. Maps::CMapLoader

```
/**
 @file MapsLoader.h

 Definición de la clase que implementa el cargador de mapas
 genérico, que lanza la activación del mapa en el código
 específico.

 @author Marco Antonio Gómez Martín
*/
#ifndef MapsLoader_H
#define MapsLoader_H

#include <string>

// Predeclaración de clases
namespace Maps {
    class CMapLoaderObserver;
}

/**
 Contiene las clases necesarias para la carga de mapas desde
 disco.
 <p>
 La implementación está hecha de tal forma que permite añadir
 soporte para nuevos formatos de mapas, de manera independiente
 al resto de la aplicación, gracias a la jerarquía de clases
 de Maps::CMapFile y Maps::CMapModel.
 <p>
 Utiliza una base de datos con todos los mapas ("físicos")
 cargados desde disco (Maps::CMapsBD), de tal forma que si se
 solicita cargar un nuevo mapa y ya ha sido cargado, no se
 realiza la carga de nuevo.
 @author Marco Antonio Gómez Martín
*/
namespace Maps {

/**
 Clase que carga y lanza un nuevo mapa desde disco.<p>
 Devuelve un valor booleano indicando si se ha podido realizar la
 carga correctamente o no. Se encarga también de informar al
 código específico que se haya registrado como observer de la
```

```
 carga del mapa. Ese código específico será avisado en el inicio
 y finalización de la carga de un mapa, así como para la creación
 de cada una de las entidades que éste contiene.

 @author Marco Antonio Gómez Martín
*/
class CMapLoader {
public:

    /**
     Carga un mapa a partir de un fichero. Si el fichero ya ha
     sido cargado previamente, no se vuelve a hacer.
     <p>
     La carga del mapa se realiza de acuerdo a su extensión, por
     lo que este método permite cargar todos aquellos tipos de
     mapas que permita CMapFile::createMapFromFile().
     <p>
     El método informa al código específico de la carga del mapa,
     para que éste inicialice/cree sus entidades (objetos de
     juego) de acuerdo a las propiedades leídas del mapa.
     <p>
     @param fileName Nombre del fichero.
     @return Si todo va bien, devuelve true; si no se puede leer
     el mapa por cualquier razón, devuelve false.
    */
    static bool loadMap(const std::string &fileName);

    /**
     Registra un nuevo observer para la carga del mapa. El
     cargador de mapas únicamente admite un observer, ya que es
     de suponer que únicamente el código específico estará
     interesado en ser informado.
     @param observer Puntero al objeto que recibirá las
     notificaciones de carga.
     @return Devuelve el puntero al objeto que anteriormente
     estaba recibiendo las notificaciones y que dejará de hacerlo
     (NULL si no existía ninguno).
     */
    inline static CMapLoaderObserver *addObserver(
                                CMapLoaderObserver *observer);


    /**
     Deregistra el observer que recibía las notificaciones de
     cargas de mapas.
     @param observer Puntero al objeto que dejará de recibir las
     notificaciones.
     @note Si el observador no estaba registrado, la operación no
     tiene ningún efecto.
     */
    inline static void removeObserver(
                                CMapLoaderObserver *observer);

private:
```

```
    /**
     Puntero al único observador registrado para ser informado de
     los eventos de carga del mapa.
     */
    static CMapLoaderObserver *_observer;
};

CMapLoaderObserver *CMapLoader::addObserver(
                                CMapLoaderObserver *observer);
    CMapLoaderObserver *antiguo = _observer;
    _observer = observer;
    return antiguo;
}

void CMapLoader::removeObserver(CMapLoaderObserver *observer) {
    if (_observer == observer)
        _observer = 0;
}

} // namespace Maps
#endif

/**
 @file MapsLoader.cpp

 Implementación de la clase del cargador de mapas genérico, que
 lanza la activación del mapa en el código específico.

 @author Marco Antonio Gómez Martín
*/
#include "MapsLoader.h"

#include "MapsLoaderObserver.h"

#include "MapFile.h"
#include "MapsBD.h"

using namespace Maps;

CMapLoaderObserver *CMapLoader::_observer = 0;

bool CMapLoader::loadMap(const std::string &fileName) {

    const CMapFile *mapFile;

    // Cargamos el mapa del disco, si no lo estaba ya
    mapFile = CMapsBD::GetPtrSingleton()
                            ->GetMapFile(fileName, true);

    if (!mapFile)
        return false;

    // Avisamos al observer de que vamos a empezar
```

```
    // la carga de un nuevo mapa
    if (_observer)
        _observer->OnBeginMapLoading(fileName);

    // Recorremos las entidades, llamando a la función de
    // construcción
    CMapFile::TListaEntidades::const_iterator it, end;

    for (it = mapFile->_listaEntidades.begin(),
         end = mapFile->_listaEntidades.end();
         it != end;
         ++it) {
        if (_observer)
            if (!_observer->InitEntity(*it))
                return false;
    }

    // Avisamos al observer de que hemos terminado
    // la carga de un nuevo mapa
    if (_observer)
        _observer->OnEndMapLoading();

    return true;
}
```

C.2. Gestor de scripts en Java

C.2.1. `JVMIntfz::JVM`

```
/**
 @file JavaIntfz.h

 Fichero que contiene el interfaz con la máquina virtual de Java.
 De esta forma, desde un programa en C++, se puede crear una JVM
 para utilizar clases implementadas en Java.

 @author Marco Antonio Gómez Martín
 */
#ifndef JAVAINTFZ_H
#define JAVAINTFZ_H

// Define este símbolo si el gestor JNI debe permitir que varias
// hebras accedan a la JVM.
#define JVMINTFZ_SUPPORT_THREAD

#include <jni.h>

#include <stdlib.h>
#ifdef _WIN32
#define PATH_SEPARATOR ';'
#else /* UNIX */
#define PATH_SEPARATOR ':'
#endif
```

```
#include <string>

#define USER_CLASSPATH "." /* where Prog.class is */

#ifdef JVMINTFZ_SUPPORT_THREAD
#include <pthread.h>
// Cuando se soportan hebras, los distintos JNIEnv se guardan
// en una lista de parejas (idHebra, JNIEnv*)
#include <utility>
#include <list>
#endif

namespace JavaIntfz {

/**
 Clase que representa la máquina virtual de Java. Debe ser
 inicializada con init(), y destruida con destroy() o llamando
 al destructor. Una JVM consume muchos recursos (de tiempo en
 la creación y de memoria durante todo el tiempo que permanece
 activa). Aún así, pueden crearse en una misma aplicación
 varias máquinas virtuales (totalmente independientes),
 utilizando dos o más objetos de la clase.

 La clase soporta la existencia de distintas hebras C++,
 Como todo este control supone una sobrecarga, el soporte o no
 para hebras es configurable a través de la macro
 JVMINTFZ_SUPPORT_THREAD.

 @author Marco Antonio Gómez
*/
class JVM {
public:
   /**
    Constructor de la clase. No crea la máquina virtual hasta
    que no se llama a init().
    */
   JVM() : jvm (NULL), idJavaLangClass(0), getNameID(0)
#ifndef JVMINTFZ_SUPPORT_THREAD
         , env(NULL)
#endif
      {};

   /**
    Destructor; elimina todos los recursos de la JVM que haya
    creados.
    */
   ~JVM() {destroy();};

   /**
    Crea la máquina virtual.
    @param classpath CLASSPATH con el que debe trabajar la JVM
      creada.
    @return true si la JVM se ha podido crear.
```

```
     */
    bool init(std::string classpath);

    /**
      Libera todos los recursos de la JVM.
    */
    void destroy();

#ifndef JVMINTFZ_SUPPORT_THREAD
    /// Registra la hebra como usuaria de JNI
    bool AttachCurrentThread() {return false;}

    /// Deregistra la hebra como usuaria de JNI
    void DettachCurrentThread() {}

    JNIEnv *operator->() {
       return env;
    }
#else
    /// Registra la hebra como usuaria de JNI
    bool AttachCurrentThread();

    /// Deregistra la hebra como usuaria de JNI
    void DettachCurrentThread();

    JNIEnv   *operator->();
#endif
    /**
     Devuelve el nombre de la clase, dado su identificador JNI.
     @param clazz
     @return Nombre de la clase Java comleto (java.util.Vector
     por ejemplo).
     */
    std::string getClassName(jclass clazz);

protected:
    /**
      Información de la máquina virtual creada con JNI.
    */
    JavaVM *jvm;

#ifndef JVMINTFZ_SUPPORT_THREAD
    /**
      Entorno de ejecución de la máquina virtual.
    */
    JNIEnv *env;
#else
    typedef std::pair<pthread_t, JNIEnv*> TInfoPorHebra;
    typedef std::list<TInfoPorHebra> TListaEntornos;
    /**
      Entornos de ejecución de la máquina virutal, uno por hebra.
     */
    TListaEntornos listaEntornos;
```

```
    /// Caché del último entorno utilizado
    TInfoPorHebra cache;

    /**
     Objeto de exclusión mutua para controlar el acceso a la
     lista de entornos y a la cache.
     */
    pthread_mutex_t mutex;
#endif

    /// Identificador de la clase java.lang.Class
    jclass idJavaLangClass;

    /// Identificador del método getName de la clase
    /// java.lang.Class
    jmethodID getNameID;
};


/**
 Convierte el nombre de una clase desde el punto de vista
 del programador de Java al nombre interpretado por
 JNI: java.util.Vector -> java/util/Vector.
 @param className Nombre de la clase original
 @return Nombre de la clase interpretable por JNI
*/
std::string getNativeClassName(const std::string &className);

}
#endif

/**
 @file JavaIntfz.cpp

 Fichero que contiene la implementación de la clase JVM, que
 permite desde un programa en C++ crear una JVM para utilizar
 clases implementadas en Java.

 @author Marco Antonio Gómez Martín
*/

#include "JavaIntfz.h"

using namespace std;
#include <assert.h>

namespace JavaIntfz {

bool JVM::init(string classpath) {

    JNIEnv *env;
    jint res;
    JavaVMInitArgs vm_args;
```

```
    // Establecemos la versión que queremos, e inicializamos
    // la estructura de los argumentos.
    vm_args.version = JNI_VERSION_1_4;
    JNI_GetDefaultJavaVMInitArgs(&vm_args);

    // Rellenamos el classpath
    // (opción -Djava.class.path=<claspath>).
    JavaVMOption options[1];

    vm_args.nOptions = 1;
    options[0].optionString = new char[classpath.length() +
                                strlen("-Djava.class.path=") + 1];
    sprintf(options[0].optionString, "-Djava.class.path=%s",
       classpath.c_str());
    vm_args.options = options;
    vm_args.ignoreUnrecognized = JNI_TRUE;

    // Creamos la JVM
    res = JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);
    delete [] (options[0].optionString);
    if (res < 0)
       return false;


    // Cargamos java.lang.Class. La utilizaremos en la
    // implementación de getClassName.
    idJavaLangClass = env->FindClass("java/lang/Class");

    if (idJavaLangClass)
       this->getNameID = env->GetMethodID(
                       idJavaLangClass,
                       "getName",
                       "()Ljava/lang/String;");

    // Guardamos el entorno creado
#ifndef JVMINTFZ_SUPPORT_THREAD
    this->env = env;
#else
    cache = TInfoPorHebra(pthread_self(), env);
    listaEntornos.push_back(cache);
    pthread_mutex_init(&mutex, NULL);
#endif

    return true;
}

void JVM::destroy() {

    if (idJavaLangClass) {
       (*this)->DeleteLocalRef(idJavaLangClass);
       idJavaLangClass = 0;
    }

#ifdef JVMINTFZ_SUPPORT_THREAD
```

```
    pthread_mutex_destroy(&mutex);
#endif

    if (jvm) {
        jvm->DestroyJavaVM();
        jvm = NULL;
    }
}

std::string JVM::getClassName(jclass clazz) {

    // La referencia a clazz la interpretamos como
    // un objeto de java.lang.Class. Llamamos al método
    // getName de ese objeto, que nos devolverá el nombre
    // de la clase.
    jobject name = (*this)->CallObjectMethod(clazz, getNameID);

    // name debería ser un java/lang/String...
    const char *cadena = (*this)->GetStringUTFChars(
                        (jstring)name, NULL);

    std::string ret(cadena);

    (*this)->ReleaseStringUTFChars((jstring)name, cadena);
    (*this)->DeleteLocalRef(name);

    return ret;

}

#ifdef JVMINTFZ_SUPPORT_THREAD

bool JVM::AttachCurrentThread() {

    pthread_mutex_lock(&mutex);

    JNIEnv *nuevoEntorno;
    int res;
    res = jvm->AttachCurrentThread((void**)&nuevoEntorno, NULL);
    if (res < 0) {
        // Error al adjuntar la hebra!
        pthread_mutex_unlock(&mutex);
        return false;
    }

    cache.first = pthread_self();
    cache.second = nuevoEntorno;
    listaEntornos.push_back(cache);

    pthread_mutex_unlock(&mutex);
    return true;
}

void JVM::DettachCurrentThread() {
```

```
    pthread_mutex_lock(&mutex);

    TListaEntornos::iterator it = listaEntornos.begin();
    TListaEntornos::iterator end = listaEntornos.end();
    // Siempre tiene que quedar la primera hebra, que es
    // la que inicializó el módulo.
    assert(it != end);
    assert(!pthread_equal(pthread_self(), it->first));

    // Buscamos la hebra
    for (++it; it != end; ++it) {
        if (pthread_equal(pthread_self(), it->first)) {
            jvm->DetachCurrentThread();
            listaEntornos.erase(it);
            pthread_mutex_unlock(&mutex);
            return;
        }
    }

    // Si hemos llegado aquí, la hebra no esta registrada.
    assert("Hebra no registrada intenta desvincularse de la JVM");

    pthread_mutex_unlock(&mutex);
}

JNIEnv *JVM::operator->() {

    pthread_mutex_lock(&mutex);

    pthread_t current;
    current = pthread_self();

    // Miramos en la caché
    if (pthread_equal(current, cache.first)) {
        JNIEnv *ret = cache.second;
        pthread_mutex_unlock(&mutex);
        return ret;
    }

    // No está; buscamos en la lista
    TListaEntornos::const_iterator it, end;

    for (it = listaEntornos.begin(), end = listaEntornos.end();
        it != end; ++it) {

        // Si la hebra es la de la lista, actualizamos la
        // caché y lo devolvemos
        if (pthread_equal(current, it->first)){
            cache = *it;
            JNIEnv *ret = cache.second;
            pthread_mutex_unlock(&mutex);
            return ret;
        }
    }
```

```
    // Si llegamos a este punto, la hebra actual NO se
    // ha encontrado como hebra registrada. Damos error!
    pthread_mutex_unlock(&mutex);
    assert("Hebra accediendo a la JVM sin haberse registrado.");

    return NULL;
}
#endif

std::string getNativeClassName(const std::string &className) {

    std::string ret(className);

    std::size_t loc = ret.find('.');
    while (loc != std::string::npos) {
        ret.replace(loc, 1, "/");
        loc = ret.find('.', loc);
    }

    return ret;
}

} // namespace
```

C.2.2. ScriptSystem::CScriptManager

```
/**
 @file ScriptManager.h

 Contiene el gestor de Scripts (en Java). El cometido inicial del
 gestor es sencillo; simplemente lanza una máquina virtual de
 Java, y admite la carga de sus clases y la llamada a métodos de
 sus objetos.

 @author Marco Antonio Gómez Martín
*/
#ifndef SCRIPT_MANAGER_H
#define SCRIPT_MANAGER_H

#include "JavaIntfz.h"
#include <map>
#include <string>
#include <vector>

#include <assert.h>

// Predeclaración de clases
namespace ScriptSystem {

class CScriptClass;
class CScriptObject;
class CStringScript;
```

```
}

// Declaración de la clase

/**
 Namespace bajo el que se encuentran las clases relacionadas con
 el control del sistmea de scripts (ejecución de Java).
 @author Marco Antonio Gómez Martín
*/
namespace ScriptSystem {

/**
 Clase gestora de todo el sistema de scripts. Es un singleton
 con inicialización y destrucción explícita, mediante Init() y
 Release().

 El sistema de scripts debe estar al tanto de las hebras que lo
 utilizan, por lo que si una hebra (que no sea la que lo
 inicializa con Init()) quiere utilizarlo, previamente tendrá que
 llamar al método AttachThread(), sin olvidarse de llamar a
 DettachThread() cuando deje de utilizarlo.

 @author Marco Antonio Gómez Martín
*/
class CScriptManager {
public:

   /**
    Método de inicialización del singleton. Debe llamarse
    al principio de la aplicación. Al final, se debe llamar
    a Release(), para liberar los recursos. Se le pasa por
    parámetro las rutas que queramos que formen el classpath,
    utilizando un vector, para que el invocante no tenga que
    conocer el carácter separador (específico de la plataforma).
    A las rutas pasadas, se añadirá la indicada en la variable
    de entorno CLASSPATH.
    @param numRutas Número de rutas distintas que formarán el
    classpath.
    @param args Array de cadenas con todas las rutas.
    @return Devuelve falso si no se ha podido inicializar
    (posiblemente será por problemas al inicializar la máquina
    virtual de Java).
    @note Esta función puede ser lenta, ya que la carga de la
    máquina virtual requiere un tiempo elevado.
   */
   static bool Init(std::vector<std::string> rutas);

   /**
    Liberación de los recursos.
   */
   static void Release();

   /**
    Indica que la hebra invocante quiere utilizar el sistema
```

```
 de scripts. Es necesario debido a exigencias de Java.
 La hebra que llama a Init() NO debe utilizarlo.
 @return Indica si se ha podido inicializar; en caso negativo,
 la hebra NO podrá utilizar el sistema de scripts, si no
 quiere generar la terminación abrupta del programa.
*/
bool AttachCurrentThread();

/**
 Solicita la desvinculación de la hebra invocante del sistema
 de scripts.
 La hebra que llamó al Init() NO debería llamar a este método.
*/
void DettachCurrentThread();

/**
 Devuelve la instancia de la clase. Debe haberse llamado
 previamente al Init().
 @return La única instancia de la clase.
 */
inline static CScriptManager *getPtrSingleton();

/**
 Devuelve el objeto que representa una clase en el sistema
 de scripts.
 @param clazz Nombre de la clase (como java.util.Vector).
 @return Puntero al objeto o NULL si no se ha podido
 acceder a la clase indicada en el parámetro.
 */
CScriptClass *getScriptClass(const std::string &nombre);

/**
 Devuelve el objeto que representa una clase en el sistema de
 scripts, a partir de un identificador nativo de la clase
 (es decir, de un identificador JNI). Ese identificador
 se ha podido obtener desde fuera utilizando algún método
 nativo JNI, y por lo tanto, es su usuario el que tendrá
 que encargarse de eliminar la referencia local.
 @param jniClassId Identificador de clase JNI.
 @return Puntero al objeto o NULL si no se ha podido acceder
 a la clase indicada en el parámetro. El invocador NO debe
 liberar el puntero; el puntero es propiedad de
 CScriptManager, y será liberado en su destructor.
 @note El CScriptManager busca en la lista de clases cargadas
 por él a ver si ya está disponible. Si no lo está, la crea
 nueva, duplicando internamente el identificador de la clase.
 Por lo tanto, en todos los casos es el usuario de este método
 el encargado de liberar con DeleteLocalRef el identificador
 pasado como parámetro.
 */
CScriptClass *getScripClassFromJniId(jclass jniClassId);

/**
 Creación de un objeto en la máquina virtual.
```

```
    El invocador deberá liberar el objeto.
    @param clazz Nombre de la clase.
    @return Referencia al objeto, o 0 si no se ha podido crear.
   */
   CScriptObject *CrearObjeto(const std::string &clazz);

   /**
    Crea un nuevo objeto de script de cadena.
    @param cad Cadena inicial
    @return Referencia al objeto, o NULL si no se pudo crear.
   */
   CStringScript *CrearCadena(const std::string &cad);

   friend class CScriptObject;
   friend class CScriptClass;
   friend class CStringScript;
protected:

   /**
    Objeto que representa la máquina virtual de Java.
    */
   JavaIntfz::JVM jvm;

   typedef std::map<std::string, CScriptClass*> TMapaClases;

   /**
    Listas de las clases Java cargadas, con su clase que la
    maneja. Los nombres de las clases (clave del map) siguen el
    formato de Java (no de JNI), como java.util.Vector, etc.
    */
   TMapaClases clasesCargadas;

   /**
   Función global que ayuda en la invocación de métodos Java.
   JNI espera en sus métodos parámetros o bien nativos
   (jboolean, etc.), o bien jobject, es decir, referencias a
   objetos. El interfaz proporcionado por ScriptSystem oculta
   toda la gestión de las  referencias a los objetos, por lo que
   los métodos para invocar métodos Java utilizan los tipos
   nativos de C++ o las clases creadas en ScriptManager
   (CScriptObject*). Esta función traduce de una lista de
   argumentos de C++ a una lista de argumentos de Java.
   @param signature Tipo de datos del método al que se invoca.
   @param cParams Lista de parámetros en C.
   @param jniParams Array de salida que se rellenará con los
   mismos parámetros pero interpretables por JNI. El tamaño del
   array se indica en el siguiente parámetro.
   @param jniParamsSize Tamaño del array anterior. Si la
   signatura indica que hay más parámetros de los que entran en
   el array, se produce un fallo.
   @return Cierto si todo fue bien; el método puede fallar por
   no ser correcta la signatura, o por tamaño del array de salida
   demasiado pequeño.
   */
```

```
    bool translateParams(const char *signature, va_list cParams,
                         jvalue *jniParams, unsigned int jniParamsSize);

private:
    /// Constructor privado (la clase es un singleton)
    CScriptManager();

    /**
    Destructor privado (la clase es un singleton)
    */
    ~CScriptManager();

    /**
     Método de construcción en dos fases.
     @param classpath Cadena con el classpath de Java que se
     pasará a la JVM que se creará.
     @return Cierto si se ha podido inicializar el objeto.
    */
    bool init(const std::string &classpath);

    /// Puntero al único objeto creado.
    static CScriptManager *_instance;
};

// Implementaciones

CScriptManager *CScriptManager::getPtrSingleton() {
    assert(_instance && "CScriptManager no inicializado!");
    return _instance;
}

} // namespace ScriptSystem

#endif
```

C.2.3. ScriptSystem::CScriptClass

```
/**
 @file ScriptClass.h

 Declaración de la clase ScriptClass, que representa una clase en
 el sistema de scripts subyacente (en la máquina virtual de Java).

 @author Marco Antonio Gómez Martín
*/
#ifndef SCRIPT_CLASS_H
#define SCRIPT_CLASS_H

#include "JavaIntfz.h"
#include <map>
#include <string>

// Predeclaración de clases
namespace ScriptSystem {
```

```
class CScriptClass;
class CScriptObject;
class CStringScript;

}

// Declaración de la clase

namespace ScriptSystem {

/**
 Clase que representa una clase en el sistema de scrips (es
 decir, una clase de la JVM). Tiene construcción en dos pasos:
 el constructor no hace nada, y el método Init() recibe como
 parámetro el nombre de la clase (Java), y devuelve cierto o
 falso dependiendo de si se ha podido inicializar.
 La construcción fuera del módulo ScriptSystem se realiza
 mediante el método factoría getScriptClass de la clase
 ScriptSystem::ScriptManager.

 @author Marco Antonio Gómez Martín
*/
class CScriptClass {
public:
   /**
   Devuelve el identificador nativo (de la JVM) de un método
   de la clase.
   @param nombre Nombre del método
   @param tipo Tipo del método según la codificación propia de
   la JVM/JNI.
   @return El identificador del método.
   */
   jmethodID getNativeMethod(const std::string &nombre,
                        const std::string &tipo) const;

   jclass getNativeClass() const {return claseJvm;}

   /**
   Creación de un objeto de la clase.
   @return Puntero al Objecto de script.
   */
   CScriptObject* CrearObjeto() const;

   // Métodos de invocación de métodos estáticos de Java
   // El parámetro name indica el nombre del método. El
   // signature su tipo, según la codificación de Java. Después
   // vienen los posibles parámetros del método.
   void InvokeStaticVoid(const char *name,
                const char *signature, ...);
   bool InvokeStaticBoolean(const char *name,
                const char *signature, ...);
   int InvokeStaticInt(const char *name,
                const char *signature, ...);
```

```
    double InvokeStaticDouble(const char *name,
                  const char *signature, ...);
    float InvokeStaticFloat(const char *name,
                  const char *signature, ...);
    CStringScript* InvokeStaticString(const char *name,
                  const char *signature, ...);
    CScriptObject* InvokeStaticObject(const char *name,
                  const char *signature, ...);

protected:

    /**
    Constructora de la clase.
    @param scriptManager Gestor de scripts utilizado.
    */
    CScriptClass(CScriptManager *sm) :
                            claseJvm(0), scriptManager(sm) {};

    /**
    Método de inicialización (inicialización en dos fases).
    @param clazz Nombre de la clase Java con el formato Java
    (no JNI), por ejemplo java.util.Vector.
    @return Cierto o falso si se ha podido inicializar.
    */
    bool Init(const std::string &clazz);

    /**
    Método de inicialización, utilizando en vez de el nombre
    de la clase, el identificador JNI. Este método solo debería
    llamarse desde dentro del propio gestor de scripts, pues de
    otra forma, no se garantiza la correcta liberación de todos
    los recursos.
    @param jniClassId Identificador de la clase en JNI.
    @return Cierto o falso si se ha podido hacer.
    */
    bool InitFromJniId(jclass jniClassId);

    /**
    Destructora de la clase. Se encarga de descargar
    la clase Java de la JVM.
    */
    ~CScriptClass();

    friend class CScriptManager;

private:

    /// Referencia a la clase en la JVM
    jclass claseJvm;

    /// Gestor de scripts bajo el que se situa la clase.
    CScriptManager *scriptManager;

    typedef std::map<std::string, jmethodID> TMapaMetodos;
```

```
    /**
    "Caché"/tabla hash con los identificadores de métodos
    obtenidos. La clave es la concatenación de nombre+signatura.
    */
    mutable TMapaMetodos metodos;

    // No permitimos la copia de objetos CScriptClass, ya que
    // requeriría la duplicación de la referencia a la clase
    // cargada de la JVM, para evitar "compartir" referencias
    // entre clases.
    CScriptClass(const CScriptClass &);
    const CScriptClass &operator=(const CScriptClass &);
};

} // namespace ScriptSystem

#endif
```

C.2.4. ScriptSystem::CScriptObject

```
/**
 @file ScriptObject.h

 Declaración de la clase ScriptObject, que representa un objeto
 en el sistema de scripts subyacente (máquina virtual de Java).

 @author Marco Antonio Gómez Martín
*/
#ifndef SCRIPT_OBJECT_H
#define SCRIPT_OBJECT_H

#include "JavaIntfz.h"
#include <map>
#include <string>

// Predeclaración de clases
namespace ScriptSystem {

class CScriptClass;
class CScriptObject;
class CStringScript;

}

// Declaración de la clase

namespace ScriptSystem {

/**
 Clase que representa un objeto en el sistema de scripts (es
 decir, un objeto de la JVM). Tiene construcción en dos pasos:
 el constructor no hace nada, y el método Init() que intenta
 construir el objeto, y devuelve cierto o falso dependiendo de
```

```
 si se ha podido o no. La construcción fuera del módulo
 ScriptSystem se realiza mediante el método factoría
 createObject de la clase ScriptSystem::CScriptManager, o
 de la clase ScriptSystem::CScriptClass.

 Las llamadas a los métodos Java se realizan utilizando la
 familia de métodos InvokeXXXX, donde el XXXX indica el tipo
 que devuelve el método invocado.

 @author Marco Antonio Gómez Martín
*/
class CScriptObject {
public:

   /**
   Destructora de la clase. Se encarga de "eliminar" el
   objeto de la JVM.
   */
   virtual ~CScriptObject();

   /**
    Comparación de dos objetos. Comprueba si los dos objetos
    representan al mismo objeto en la JVM.
    Se necesita sobrecargar ya que las referencias JNI
    pueden ser distintas, y sin embargo, representar lo mismo.
    */
   bool operator==(const CScriptObject &);

   // Métodos de invocación de métodos Java
   // El parámetro name indica el nombre del método. El
   // signature su tipo, según la codificación de Java. Después
   // vienen los posibles parámetros del método.

   void InvokeVoid(const char *name,
                   const char *signature, ...);
   bool InvokeBoolean(const char *name,
                   const char *signature, ...);
   int InvokeInt(const char *name,
                   const char *signature, ...);
   double InvokeDouble(const char *name,
                   const char *signature, ...);
   float InvokeFloat(const char *name,
                   const char *signature, ...);
   CStringScript* InvokeString(const char *name,
                   const char *signature, ...);
   CScriptObject* InvokeObject(const char *name,
                   const char *signature, ...);

   friend class CScriptClass;
protected:

   /**
   Constructora de la clase.
   @param jvmIntfz Puntero al objeto que representa la JVM
```

```
    */
    CScriptObject(CScriptManager *sm) :
                            scriptManager(sm), objJvm(0) {};

    /**
     Factoría de la clase, construye un objeto a partir de una
     referencia JNI. El objeto se preocupará, en su destructor, de
     liberar la referencia al objeto de la JVM. El usuario será el
     responsable de liberar el objeto.
     @param sm ScriptManager que se está utilizando
     @param obj Objeto JNI
     */
    static CScriptObject *createFromJNIObject(CScriptManager *sm,
                    jobject obj);

    /**
    Método de inicialización (inicialización en dos fases).
    @param clase Clase de la que se creará el objeto.
    @return Cierto o falso si se ha podido inicializar.
    */
    bool Init(const CScriptClass &clase);

    /**
    Método de inicialización (inicialización en dos fases),
    utilizando en vez de el nombre de la clase, el identificador
    JNI (el objeto se supone ya construido, es decir, no se
    llamará a su constructor).
    @param jniObjectId Identificador del objeto en JNI.
    @return Cierto o falso si se ha podido hacer.
    */
    bool InitFromJniId(jobject jniObjectId);

    friend class CScriptManager;

    /// Referencia al objeto en la JVM
    jobject objJvm;

    /// Puntero al script manager a utilizar
    CScriptManager *scriptManager;

    /// Clase a la que pertenece el objeto
    const CScriptClass *clazz;

    // No permitimos la copia de objetos CScriptObject, ya que
    // requeriría la duplicación de la referencia al objeto
    // cargado de la JVM, para evitar "compartir" referencias.
    CScriptObject(const CScriptObject &);
    const CScriptObject &operator=(const CScriptObject &);
};

} // namespace ScriptSystem

#endif
```

C.3. Máquina virtual de Java

C.3.1. JJVM::COperandStack

```
/**
 @file OperandStack.h

 Definición de la clase que implementa la pila de operandos
 de la JVM.

 @author Marco Antonio Gómez Martín y Pedro Pablo Gómez Martín
*/
#ifndef JJVM_OperandStackH
#define JJVM_OperandStackH

#include <vector>
#include <list>

#include "JVMTypes.h"

namespace JJVM {

/**
 La pila de operandos almacena los operandos de algunas de las
 instrucciones de la JVM (iadd, etc.). Cada entrada puede guardar
 un valor de cualquier tipo de la JVM, incluyendo long y double.
 (Estos dos tipos contribuyen en dos unidades en la profundidad
 de la pila).

 Los métodos de la clase no comprueban errores, como pila vacía.

 Aunque la clase tiene métodos para funcionar como una pila,
 también permite el acceso al resto de elementos.

 También permite registrar observadores de los cambios.

 @author Marco Antonio Gómez Martín, Pedro Pablo Gómez Martín
*/
class COperandStack {
public:

   /// Constructor
   inline COperandStack();

   /// Destructor
   ~COperandStack() {};

   /**
    Devuelve la profundidad de la pila. Los tipos long y double
    ocupan dos unidades.
    @return Profundidad de la pila.
   */
   inline int getSize() const;
```

```
/**
 Devuelve el tipo del objeto en la cima.
 @return Una constante que indica el tipo (JJVM\JVMTypes.h)
*/
inline JVMType getTypeInTop() const;

/**
 Devuelve el tipo del objeto en la posición indicada.
 @param pos Posición en la pila del valor. Si el tipo utiliza
 dos posiciones, debe utilizarse la mayor.
 * @return Una constante que indica el tipo (JJVM\JVMTypes.h)
 */
inline JVMType getType(int pos) const;

/**
 * Devuelve y borra el entero de la cima de la pila.
 * @return Entero en la cima.
 */
jint popInt();

// Se omiten los métodos para los tipos long,
// float, double, returnAddress y jref

// Métodos para acceder a elementos de la pila. Si éstos
// ocupan más de una posición, hay que utilizar el más alto.
jint         getInt(int pos) const;
jlong        getLong(int pos) const;
jfloat       getFloat(int pos) const;
jdouble      getDouble(int pos) const;
jretAddr     getReturnAddress(int pos) const;
jref         getReference(int pos) const;

inline jint topInt() const {
   return getInt(getSize()-1);
}
inline jlong topLong() const {
   return getLong(getSize()-1);
}
inline jfloat topFloat() const {
   return getFloat(getSize()-1);
}
inline jdouble topDouble() const {
   return getDouble(getSize()-1);
}
inline jretAddr topReturnAddress() const {
   return getReturnAddress(getSize()-1);
}
inline jref topReference() const {
   return getReference(getSize()-1);
}

void      pushInt(jint);
void      pushLong(jlong);
```

```
void      pushFloat(jfloat);
void      pushDouble(jdouble);
void      pushReturnAddress(jretAddr);
void      pushReference(jref);


/**
 Iterador que permite recorrer todos los elementos de la
 pila, teniendo en cuenta los distintos tamaños de sus
 valores. El iterador no soporta cambios en la pila a mitad
 del recorrido.

 Ejemplo de uso:
 it = stack.getIteratorUpDown();
 while (!it.end()) {
    stack.getType(it);
    ...
    it++;
 }
*/
class CIterator {
public:
   CIterator() {}
   CIterator& operator++(int);
   operator int() const { return cont; }
   bool end();
protected:
   COperandStack *stack;
   bool up2down;
   int cont;

   friend class COperandStack;
}; // class CIterator

CIterator getIteratorUpDown();
CIterator getIteratorDownUp();

/**
 Interface observador de cambios de la pila de operandos.
*/
class Observer {
public:
   /**
    * Función llamada después de un push.
    * @param stack Pila cambiada.
    */
   virtual void push(const COperandStack &stack) = 0;

   /**
    * Función llamada después de un pop.
    * @param stack Pila cambiada.
    */
   virtual void pop(const COperandStack &stack) = 0;
}; // class Observer
```

```
    /**
     Añade un nuevo observer de la pila.
     @param newObserver Nuevo observador.
     @return true si todo fue bien.
    */
    bool addObserver (Observer* newObserver);

    /**
     Elimina un observer.
     @param oldObserver Observer que será eliminado.
     @return true si todo fue bien.
    */
    bool removeObserver (Observer* oldObserver);

protected:

    struct   OperandContent {
       jint  cont;
       JVMType   type;
    };

    void  push(char *data, int size, JVMType type);
    void  get(char *data, int size, int pos) const;

    /// Envía el evento "push" a todos los observers
    void emitPush() const;

    /// Envía el evento "pop" a todos los observers
    void emitPop() const;

    int   length;
    std::vector<OperandContent>  stack;

    std::list </*COperandStack::*/Observer*> observers;

};

COperandStack::COperandStack() : length(0) {
}

int   COperandStack::getSize() const {
    return length;
}

JVMType COperandStack::getTypeInTop() const {
    return stack[length-1].type;
}

JVMType COperandStack::getType(int pos) const {
    if ((0 <= pos) || (pos < length))
       return stack[pos].type;
    return JJVM::TYPE_ERROR;
}
```

```
} // namespace JJVM

#endif

/**
 @file OperandStack.cpp

 Implementación de la clase que implementa la pila de operandos
 de la JVM.

 @author Marco Antonio Gómez Martín y Pedro Pablo Gómez Martín
*/

#include "OperandStack.h"
#include "JVMTypes.h"

#include <memory>       // memcpy(...)

using namespace JJVM;

//-------------------------------------------------------------------
// MÉTODOS POP
//-------------------------------------------------------------------
jint COperandStack::popInt() {
   jint d;
   d = getInt(length - 1);
   length--;
   stack.pop_back();
   emitPop();
   return d;
}
jlong COperandStack::popLong() {
   jlong d;
   d = getLong(length - 1);
   length-=2;
   stack.pop_back();
   stack.pop_back();
   emitPop();
   return d;
}

// Omitidos popFloat, popDouble, popReturnAddress y popReference

//-------------------------------------------------------------------
// MÉTODOS GET
//-------------------------------------------------------------------
jint COperandStack::getInt(int pos) const {
   jint d;
   get((char*)&d, sizeof(d), pos);
   return d;
}

// Omitidos getLong, getFloat, getDouble, getReturnAddress
```

```
// y getReference

//-------------------------------------------------------------------
// MÉTODOS PUSH
//-------------------------------------------------------------------

void COperandStack::pushInt(jint d) {
   push((char*)&d, sizeof(d), TYPE_INT);
}

// Omitidos pushLong, pushFloat, pushDouble, pushReturnAddress
// y pushReference

//-------------------------------------------------------------------
// OTROS MÉTODOS
//-------------------------------------------------------------------

COperandStack::CIterator COperandStack::getIteratorUpDown() {

   CIterator it;
   it.stack = this;
   it.up2down = true;
   it.cont = length - 1;
   return it;

} // getIteratorUpDown

COperandStack::CIterator COperandStack::getIteratorDownUp() {

   CIterator it;
   it.stack = this;
   it.up2down = false;
   if (length == 0)
      it.cont = -1;
   else if (getTypeSize(getType(0)) == 8)
      it.cont = 1;
   else
      it.cont = 0;

   return it;

} // getIteratorDownUp

bool COperandStack::addObserver (Observer* newObserver) {

   observers.push_back(newObserver);
   return true;

} // addObserver

bool COperandStack::removeObserver (Observer* oldObserver) {

   observers.remove(oldObserver);
```

```
    return true;

} // removeObserver

//-------------------------------------------------------------------
// MÉTODOS PROTEGIDOS
//-------------------------------------------------------------------

void COperandStack::push(char *data, int size, JVMType type) {

    while (size != 0) {
        OperandContent aux;

        int    toCopy = size > sizeof(aux.cont) ?
          sizeof(aux.cont) : size;

        memcpy((char*)&aux.cont, data, toCopy);
        aux.type = type;
        stack.push_back(aux);
        length++;
        size -= toCopy;
        data += toCopy;
    }

    emitPush();
}

void COperandStack::get(char *data, int size, int pos) const {

    if ((0 <= pos) && (pos < length)) {

        int    numPos = (size-1)/4;
        pos -= numPos;

        while (size != 0) {
            OperandContent aux = stack[pos];

            int    toCopy = size > sizeof(aux.cont) ?
              sizeof(aux.cont) : size;
            memcpy(data, (char*)&aux.cont, toCopy);
            data += toCopy;
            size -= toCopy;
            pos++;
        }
    }

} // get

void COperandStack::emitPush() const {

    std::list<COperandStack::Observer*>::const_iterator it =
      observers.begin();

    while (it != observers.end()) {
```

```
      (*it)->push(*this);
      ++it;
   }

} // emitPush

void COperandStack::emitPop() const {
   std::list<COperandStack::Observer*>::const_iterator it =
     observers.begin();

   while (it != observers.end()) {
      (*it)->pop(*this);
      ++it;
   }

} // emitPop

//-----------------------------------------------------------
// Implementación COperandStack::CIterator
//-----------------------------------------------------------

COperandStack::CIterator&
COperandStack::CIterator::operator++(int) {
   if (!end()) {
      if (up2down)
         cont -= JJVM::getTypeSize(stack->getType(cont)) / 4;
      else {
         cont++;
         if (!end() &&
             (JJVM::getTypeSize(stack->getType(cont)) == 8))
            cont++;
      }
   }
   return *this;
}

bool COperandStack::CIterator::end() {
   if (up2down)
      return cont == -1;
   else
      return cont >= stack->getSize();
}
```

Bibliografía

Como un ganso desplumado y escuálido,
me preguntaba a mí mismo con voz indecisa
si de todo lo que estaba leyendo
haría el menor uso alguna vez en la vida.

James Clerk Maxwell, sobre su educación en Cambridge

Abbattista, F., Degemmis, M., Fanizzi, N., Licchelli, O., Lops, O., Semeraro, G. y Zambetta, F. Learning user profiles for content-based filtering in e-commerce. En *Workshop Apprendimento Automatico: Metodi e Applicazioni*. Sienna, Italia, 2002.

Abrash, M. *Michael Abrash's Graphics Programming Black Book (Special Edition)*. Coriolis Group Books, 1997. ISBN 1-576-10174-6.

Adobe. Macromedia authorware 7. 2006. Disponible en `http://www.adobe.com/products/authorware/` (último acceso, Agosto, 2007).

Adolph, S. Whatever happened to reuse? *Software Development magazine*, 1999. Disponible en `http://www.ddj.com/dept/architect/184415752` (último acceso, Agosto, 2007).

Advanced Distributed Learning (ADL), editor. *Sharable Content Object Reference Model (SCORM) 2004, 2nd Edition, Overview*. ADL Co-Laboratory, 2004. Disponible en `http://www.adlnet.gov/` (último acceso, Agosto, 2007).

Agile Alliance. Manifesto for agile software development. 2001. Disponible en `http://agilemanifesto.org` (último acceso, Agosto, 2007).

Alderman, D. L. Evaluation of the TICCIT computer-assisted instructional system in the community college. *ACM SIGCUE Bulletin*, vol. 13(3), páginas 5–17, 1979. ISSN 0163–5735.

Alexandrescu, A. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, 2001. ISBN 0-20170-431-5.

AMSOFT. El laberinto del sultan (título original, Sultan's maze). 1984.

ANASTASSAKIS, G., PANAYIOTOPOULOS, T. y RITCHINGS, T. Virtual agent societies with the mVITAL intelligent agent system. En *Proceedings of the Third International Workshop on Intelligent Virtual Agents (IVA'01)* (editado por A. de Antonio, R. Aylett y D. Ballin), vol. 2190 de *Lecture Notes in Artificial Intelligence (subserie de LNCS)*, páginas 210–223. Springer-Verlag, 2001a.

ANASTASSAKIS, G., RITCHINGS, T. y PANAYIOTOPOULOS, T. Multi–agent systems as intelligent virtual environments. En *Proceedings of the Joint German/Austrian Conference on Artificial Intelligence* (editado por F. Baader, G. Brewka y T. Eiter), vol. 2174 de *Lecture Notes in Computer Science*, páginas 381–395. Springer-Verlag, 2001b.

DE ANTONIO, A., RAMÍREZ, J., IMBERT, R. y MÉNDEZ, G. Intelligent virtual environments for training: An agent-based approach. En *4th International Central and Eastern European Conference on Multi-Agent Systems* (editado por M. Pechoucek, P. Petta y L. Z. Varga), vol. 3690 de *Lecture Notes in Computer Science*, páginas 82–91. Springer-Verlag, 2005. ISBN 3-540-29046-X.

DE ANTONIO, A., RAMÍREZ, J., IMBERT, R., MÉNDEZ, G. y AGUILAR, R. A. A software architecture for intelligent virtual environments applied to education. *Revista de la Facultad de Ingeniería, Universidad de Taracapá*, vol. 13(1), 2004. ISSN 0717-1072.

ARÉVALO, J. Main loop with fixed time steps. Publicado por Flipcode, 2001. Disponible en `http://www.flipcode.org/cgi-bin/fcarticles.cgi?show=63823` (último acceso, Octubre, 2006).

ARÉVALO, J. Videojuegos: Arte, diversión, ingeniería, negocio. Charla impartida en la Facultad de Informática de la Universidad Comlutense de Madrid, 2005. Disponible en `http://www.iguanademos.com/Jare/docs/2005UCMVideojuegos.zip` (último acceso, Agosto, 2007).

ARÉVALO, J. Desarrollo de videojuegos. Charla impartida en la Universidad Politécnica de Cataluña, 2006. Disponible en `http://www.iguanademos.com/Jare/docs/UPC2006-DesarrolloDeVideojuegos.ppt` (último acceso, Agosto, 2007).

ASTRACHAN, O., MITCHENER, G., BERRY, G. y COX, L. Design patterns: an essential component of CS curricula. En *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education (SIGCSE'98)*, páginas 153–160. ACM Press, New York, NY, USA, 1998. ISBN 0-89791-994-7.

AYLETT, R. y CAVAZZA, M. Intelligent virtual environments - A state-of-the-art report. En *Eurographics Conference*. Manchester, UK, 2001.

AYLETT, R. y LUCK, M. Applying artificial intelligence to virtual reality: Intelligent virtual environments. *Applied Artificial Intelligence*, vol. 14(1), páginas 3–32, 2000.

BARES, W. H., ZETTLEMOYER, L. S. y LESTER, J. C. Habitable 3D learning environments for situated learning. En *Proceedings of the 4th International Conference on Intelligent Tutoring Systems (ITS'98)*, páginas 76–85. Springer-Verlag, London, UK, 1998. ISBN 3-540-64770-8.

BASS, L., CLEMENTS, P. y KAZMAN, R., editores. *Software Architecture in Practice*. Addison–Wesley, 2003.

BELLIFEMINE, F., POGGI, A. y RIMASSA, G. Developing multi-agent systems with a FIPA-compliant agent framework. *Software - Practice And Experience*, vol. 31(2), páginas 103–128, 2001a.

BELLIFEMINE, F., POGGI, A. y RIMASSA, G. JADE: a FIPA2000 compliant agent development environment. En *Proceedings of the fifth international conference on Autonomous agents*, páginas 216–217. ACM Press, 2001b. ISBN 1-58113-326-X.

BILAS, S. *Game Programming Gems*, capítulo A generic function-binding interface, páginas 56–67. Charles River Media, 2000. ISBN 1-58450-049-2.

BIOWARE-CORP. NeverWinter Nights + Shadow of Undrentide (expansion pack). 2002.

BIRKHOFF, G. *Lattice Theory*. American Mathematical Society, tercera edición, 1973. ISBN 0-821-81025-1.

DU BOUCHER-RYAN, P. y BRIDGE, D. Collaborative recommending using formal concept analysis. En *AI-2005, the XXV SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence* (editado por M. Bramer, F. Coenen y T. Allen), páginas 205–218. Springer, Cambridge, UK, 2005. ISBN 978-1-84628-225-6.

BOURG, D. M. y SEEMANE, G. *AI for Game Developers*, capítulo Rule-Based AI, páginas 212–227. Charles River Media, 2004. ISBN 0-596-00555-5.

BRAUDE, E. J. *Software Engineering. An Object-Oriented Perspective*. John Wiley & Sons, Inc., 2001.

BROCK, K. Mensaje en sweng-gamedev: Strategies/patterns for composition. 2006. Disponible en `http://lists.midnightryder.`

com/pipermail/sweng-gamedev-midnightryder.com/2006-October/006087.html (último acceso, Noviembre, 2006).

BRUSILOVSKY, P. Student model centered architecture for intelligent learning environment. En *4th International Conference on User Modeling*, páginas 31–36. Hyannis, MA, USA, 1994.

BRUSILOVSKY, P. Intelligent learning environments for programming: The case for integration and adaptation. En *Proceedings of the 7th World Conference on Artificial Intelligence in Education (AI-ED'95)* (editado por J. Greer), páginas 1–8. 1995.

BRUSILOVSKY, P. Adaptive hypermedia. *User Modeling and User-Adapted Interaction*, vol. 11, páginas 81–110, 2001.

BURTON, R. R. y BROWN, J. S. An investigation of computer coaching for informal learning activities. En *Intelligent Tutoring Systems* (editado por D. Sleeman y J. S. Brown), páginas 79–98. Academic Press, London, England, 1982.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., STAL, M., SOMMERLAD, P. y STAL, M. *Pattern-Oriented Software Architecture*, vol. 1: A System of Patterns. John Wiley and Sons, primera edición, 1996.

CARBONELL, J. R. AI in CAI: an artificial intelligence approach to computer-assisted instruction. *IEEE Transactions on Man-Machine Systems*, vol. 11(4), páginas 190–202, 1970.

CARPINETO, C. y ROMANO, G. A lattice conceptual clustering system and its application to browsing retrieval. *Machine Learning*, vol. 24(2), páginas 95–122, 1996.

CASSELL, J., BICKMORE, T. W., CAMPBELL, L., VILHJÁLMSSON, H. H. y YAN, H. More than just a pretty face: conversational protocols and the affordances of embodiment. *Knowledge-Based Systems*, vol. 14(1-2), páginas 55–64, 2001.

CHARLES, D. y BLACK, M. Dynamic player modelling: A framework for player-centric digital games. En *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*, páginas 29–35. 2004.

CHARLES, D., MCNEILL, M., MCALISTER, M., BLACK, M., MOORE, A., STRINGER, K., KÜCKLICH, J. y KERR, A. Player-centred game design: Player modelling and adaptive digital games. En *Proocedings of Digital Games Research Association (DIGRA) Conference: Changing Views – Worlds in Play*. 2005.

CHITTARO, L. y RANON, R. Dynamic generation of personalized VRML content: a general approach and its application to 3D e-commerce. En *Proceeding of the seventh international conference on 3D Web technology*, páginas 145–154. ACM Press, 2002. ISBN 1-58113-468-1.

COLLINS-SUSSMAN, B., FITZPATRICK, B. W. y PILATO, C. M. *Version Control with Subversion*. O'Reilly, 2004. ISBN 0-596-00448-6.

COMBS, N. y ARDOINT, J.-L. Declarative versus imperative paradigms in games AI. 2005. Disponible en `http://www.roaringshrimp.com/WS04-04NCombs.pdf` (último acceso, Agosto, 2007).

CRAWFORD, C. *The Art of Computer Game Design*. Mcgraw-Hill, 1984. ISBN 0-078-81117-1. Disponible en `http://www.vancouver.wsu.edu/fac/peabody/game-book/Coverpage.html` (último acceso, Agosto, 2007).

CRAWFORD, C. *Chris Crawford on Game Design*. New Riders Games, 2003. ISBN 0-131-46099-4.

CRYTEK. *CryEngine Sandbox - Far Cry Edition*. Crytek GmbH, 2004.

CRYTEK STUDIOS. Far Cry. 2004.

DAVEY, B. A. y PRIESTLEY, H. A. *Introduction to Lattices and Order*. Cambridge University Press, segunda edición, 2002. ISBN 0-521-36766-2.

DAVIS, R. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, vol. 24(1-3), páginas 347–410, 1984. ISSN 0004–3702.

DAWSON, B. *Game Programming Gems 2*, capítulo Micro-Threads for Game Object AI, páginas 258–254. Charles River Media, 2001. ISBN 1-584-50054-9.

DELOURA, M., editor. *Game Programming Gems*. Charles River Media, 2000. ISBN 1-584-50049-2.

DELOURA, M., editor. *Game Programming Gems 2*. Charles River Media, 2001. ISBN 1-584-50054-9.

DEVEDZIC, V. y HARRER, A. Software patterns in ITS architectures. *International Journal of Artificial Intelligence in Education*, vol. 15, páginas 63–94, 2005. ISSN 1560–4306.

DEVEDZIC, V., RADOVIC, D. y JERINIC, L. On the notion of components for intelligent tutoring systems. En *4th International Conference on Intelligent Tutoring Systems (ITS'98)* (editado por B. P. Goettl, H. M. Halff, C. L. Redfield y V. J. Shute), vol. 1452 de *Lecture Notes in Computer Science*, páginas 504–513. Springer-Verlag, 1998. ISBN 3-540-64770-8.

DHAR, V. y POPLE, H. E. Rule-based versus structure-based models for explaining and generating expert behavior. *Communication ACM*, vol. 30(6), páginas 542–555, 1987. ISSN 0001–0782.

DÍAZ AGUDO, B., GÓMEZ MARTÍN, M. A., GÓMEZ MARTÍN, P. P. y GONZÁLEZ CALERO, P. A. Formal concept analysis for knowledge refinement in case based reasoning. En *AI-2005, the XXV SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence* (editado por M. Bramer, F. Coenen y T. Allen), páginas 233–245. Springer, 2005. ISBN 978-1-84628-225-6.

DICKHEISER, M., editor. *Game Programming Gems 6*. Charles River Media, 2006. ISBN 1-584-50450-1.

DIJKSTRA, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik*, vol. 1, páginas 269–271, 1959.

DINAMIC. Don quijote. 1987.

DOUGIAMAS, M. y TAYLOR, P. Moodle: Using learning communities to create an open source course management system. En *World Conference on Educational Multimedia, Hypermedia and Telecommunications* (editado por P. Kommers y G. Richards), páginas 171–178. 2003.

DUFFY, J. Postcard from the Montreal game summit: Call of duty 2 postmortem. 2005. Disponible en `http://www.gamasutra.com/features/20051104/duffy_01.shtml` (último acceso, Agosto, 2007).

DUQUENNE, V. y GUIGUES, J. Famille minimale dímplication informatives résultant dún tableau de données binaries. *Mathématiques et Sciences Humaines*, vol. 24(95), 1986.

DYBSAND, E. *AI Game Programming Wisdom*, capítulo A Finite-State Machine Class. Charles River Media, 2002. ISBN 1-584-50077-8.

EBERLY, D. H. *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*. Morgan Kaufmann, 2000. ISBN 1-558-605932.

EBERLY, D. H. *3D Game Engine Architecture: Engineering Real-Time Applications with Wild Magic*. Morgan Kaufmann, 2004. ISBN 0-122-29064X.

ECKEL, B. *Piensa en Java*, capítulo 12, Identificación de tipos en tiempo de ejecución. Prentice Hall, 2003.

EISENBARTH, T., KOSCHKE, R. y SIMON, D. Locating features in source code. *IEEE Transactions on Software Engineering*, vol. 29(3), páginas 210–224, 2003.

ELADHARI, M. *Object Oriented Story Construction in Story Driven Computer Games*. Proyecto Fin de Carrera, Department of History of Literature and History of Ideas, Stockholm University, 2002.

ENSEMBLE-STUDIOS. Age of empires II: The age of king. 1999. Disponible en `http://www.microsoft.com/latam/juegos/Age2/default.asp` (último acceso, Marzo, 2007).

EPIC GAMES. Unreal tournament. 1999.

DE FIGUEIREDO, L. H., CELES, W. y IERUSALIMSCHY, R. *Game Programming Gems 6*, capítulo Programming advanced control mechanisms with Lua coroutines, páginas 357–370. Charles River Media, 2006. ISBN 1-584-50450-1.

FINKEL, R. A. y BENTLEY, J. L. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, vol. 4(1), páginas 1–9, 1974. ISSN 0001-5903.

FLETCHER, J. D. Modeling of learner in computer-based instruction. *Journal of Computer-Based Instruction*, vol. 1, páginas 118–126, 1975.

FOLEY, J. D., VAN DAM, A., FEINER, S. K. y HUGHES, J. F. *Computer Graphics: Principles and Practice*. Addison-Wesley, 1990. ISBN 0-201-12110-7.

FOWLER, M. Continuous integration. 2006. Disponible en `http://www.martinfowler.com/articles/continuousIntegration.html` (último acceso, Agosto, 2007).

FRANKLIN, S. y GRAESSER, A. Is it an agent, or just a program? A taxonomy for autonomous agents. En *Third International Workshop on Agent Theories, Architectures and Languages*, vol. 1193 de *Lecture Notes in Computer Science*, páginas 21–35. Springer, 1996. ISBN 3-540-62507-0.

FRIEDMAN-HILL, E. JESS, the rule engine for the Java platform. 2005. Disponible en `http://herzberg.ca.sandia.gov/jess/docs/61/` (último acceso, Marzo, 2007).

FRISTROM, J. Manager in a strange land: Most projects suck. 2003. Disponible en `http://www.gamasutra.com/features/20031017/fristrom_pfv.htm` (último acceso, Agosto, 2007).

GABB, H. y LAKE, A. Threading 3D game engine basics. 2005. Disponible en `http://www.gamasutra.com/features/20051117/gabb_01.shtml` (último acceso, Agosto, 2007).

GAMMA, E., HELM, R., JOHNSON, R. y VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, primera edición, 1995.

GANTER, B. y WILLE, R. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1998. ISBN 3-540-62771-5.

GARCÉS, S. *AI Game Programming Wisdom III*, capítulo Flexible Object-Composition Architecture. Charles River Media, 2006. ISBN 1-584-50457-9.

GIARRATANO, J. C. CLIPS user's guide. 2002. Disponible en `http://www.ghg.net/clips/download/documentation/usrguide.pdf` (último acceso, Agosto, 2007).

GILLEN, K. The first generation of the next generation: A project Gotham racing 3 postmortem. 2005. Disponible en `http://www.gamasutra.com/features/20050906/gillen_01.shtml` (último acceso, Agosto, 2007).

GIRAFFA, L. M. M. y VICCARI, R. M. The use of agents techniques on intelligent tutoring systems. En *XVIII International Conference of the Chilean Computer Science Society*, páginas 76–83. IEEE Computer Society, Los Alamitos, CA, USA, 1998. ISSN 1522-4902.

GÖKER, M. H. Adapting to the level of experience of the user in mixed-initiative web self-service applications. En *Workshop on Mixed Initiative Case-Based Reasoning, at the 5th International Conference on Case Based Reasoning (ICCBR'03)* (editado por D. Aha). 2003.

GÖKER, M. H. y THOMPSON, C. A. Personalized conversational case-based recommendation. En *Advances in Case-Based Reasoning. Proceedings of the 8th European Workshop on Case-Based Reasoning (EWCBR'00)* (editado por E. Blanzieri y L. Protinale), vol. 1898 de *Lecture Notes in Artificial Intelligence (subserie de LNCS)*. Springer, 2000. ISBN 3-540-67933-2.

GÓMEZ GAUCHÍA, H., DÍAZ AGUDO, B. y GONZÁLEZ CALERO, P. A. Automatic personalization of the human computer interaction using temperaments. En *Proceedings of the 15th Int. Florida Artificial Intelligence Research Society Conference (FLAIRS'06)*. AAAI Press, Florida, USA, 2006.

GÓMEZ MARTÍN, M. A. y DÍAZ AGUDO, B. Evaluación de estrategias de razonamiento para sistemas basados en reglas. En *JENUI'06: XII Jornadas de Enseñanza Universitaria de la Informática*, páginas 303–310. Editorial Thomson, 2006. ISBN 84-9732-545-1.

GÓMEZ MARTÍN, M. A. y GÓMEZ MARTÍN, P. P. Uso de aplicaciones de ejemplo en las clases de informática gráfica. En *JENUI'05: XI Jornadas de Enseñanza Universitaria de la Informática*, páginas 421–428. Editorial Thomson, 2005. ISBN 84-9732-421-8.

GÓMEZ MARTÍN, M. A., GÓMEZ MARTÍN, P. P. y GONZÁLEZ CALERO, P. A. Aprendizaje basado en juegos. *La Revista Icono 14*, (4), 2004a. ISSN 1697-8293.

GÓMEZ MARTÍN, M. A., GÓMEZ MARTÍN, P. P. y GONZÁLEZ CALERO, P. A. Game-driven intelligent tutoring systems. En *Entertainment Computing, Third International Conference (ICEC'04)* (editado por M. Rauterberg), vol. 3166 de *Lecture Notes in Computer Science*, páginas 108–113. Springer, 2004b. ISBN 3-540-22947-7.

GÓMEZ MARTÍN, M. A., GÓMEZ MARTÍN, P. P. y GONZÁLEZ CALERO, P. A. Game-based learning as a new domain for case-based reasoning. En *1st Workshop on Computer Gaming and Simulation Environments, at 6th International Conference on Case-Based Reasoning (ICCBR)* (editado por D. W. Aha y D. Wilson), páginas 175–184. Chicago, IL, US, 2005a.

GÓMEZ MARTÍN, M. A., GÓMEZ MARTÍN, P. P. y GONZÁLEZ CALERO, P. A. Dynamic binding is the name of the game. En *Entertainment Computing (ICEC 2006), 5th International Conference* (editado por R. Harper, M. Rauterberg y M. Combetto), vol. 4161 de *Lecture Notes in Computer Science*, páginas 229–232. Springer, 2006a. ISBN 978-3-540-45259-1. ISSN 0302-9743.

GÓMEZ MARTÍN, M. A., GÓMEZ MARTÍN, P. P. y GONZÁLEZ CALERO, P. A. Aprendizaje activo en simulaciones interactivas. *Revista Iberoamericana de Inteligencia Artificial*, vol. 11(33), páginas 25–36, 2007a.

GÓMEZ MARTÍN, M. A., GÓMEZ MARTÍN, P. P. y GONZÁLEZ CALERO, P. A. Data driven software architecture for game-based learning systems. En *Learning with Games* (editado por M. Taisch y J. Cassina), páginas 229–236. 2007b. ISBN 978-88-901168-0-3.

GÓMEZ MARTÍN, M. A., GÓMEZ MARTÍN, P. P., GONZÁLEZ CALERO, P. A. y DÍAZ AGUDO, B. Adjusting game difficulty level through formal concept analysis. En *AI-2006, the XXVI SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence* (editado por M. Bramer, F. Coenen y T. Allen), páginas 217–230. Springer, 2006b. ISBN 978-1-84628-662-9.

GÓMEZ MARTÍN, M. A., GÓMEZ MARTÍN, P. P., GONZÁLEZ CALERO, P. A. y JIMÉNEZ DÍAZ, G. JV^{2}M, un sistema de enseñanza de la compilación de Java. En *I Simposio Nacional de Tecnologías de la Información*

y de las Comunicaciones en la Educación, SINTICE 2005 (editado por M. Ortega Cantero), páginas 259–266. Thomson, 2005b. ISBN 84-9732-437-4.

GÓMEZ MARTÍN, M. A., GÓMEZ MARTÍN, P. P. y JIMÉNEZ DÍAZ, G. Integrating a support tool infrastructure in development courses. En *9th International Symposium on Computers in Education (SIIE'07)*. Pendiente de publicación, 2007c.

GÓMEZ MARTÍN, M. A., GÓMEZ MARTÍN, P. P., PALMIER CAMPOS, P. y GONZÁLEZ CALERO, P. A. Not yet another visualization tool: Learning compilers for fun. En *8th International Symposium on Computers in Education (SIIE'06)* (editado por L. Panizo Alonso, L. Sánchez González, B. Fernández Manjón y M. Llamas Nistal), páginas 264–271. Universidad de León, León, Spain, 2006c. ISBN 84-9773-301-0.

GÓMEZ MARTÍN, P. P. *Modelo de enseñanza basado en casos: de los tutores inteligentes a los videojuegos*. Tesis Doctoral, Facultad de Informática, Universidad Complutense de Madrid, 2007.

GÓMEZ MARTÍN, P. P. y GÓMEZ MARTÍN, M. A. Fast application development to demonstrate computer graphics concepts. En *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE'06)*, páginas 250–254. ACM Press, New York, USA, 2006. ISBN 1-59593-055-8.

GÓMEZ MARTÍN, P. P., GÓMEZ MARTÍN, M. A., DÍAZ AGUDO, B. y GONZÁLEZ CALERO, P. A. Opportunities for CBR in learning by doing. En *Proceedings of Case-Based Reasoning Research and Development, 6th International Conference on Case-Based Reasoning, ICCBR 2005* (editado por H. Muñoz Avila y F. Ricci), vol. 3620 de *Lecture Notes in Artificial Intelligence, subseries of LNCS*, páginas 267–281. Springer, Chicago, IL, US, 2005c. ISBN 978-3-540-28174-0.

GÓMEZ MARTÍN, P. P., GÓMEZ MARTÍN, M. A. y GONZÁLEZ CALERO, P. A. JAVY: Virtual Environment for Case-Based Teaching of Java Virtual Machine. En *7th International Conference, Knowledge-Based Intelligent Information and Engineering Systems (KES 2003)*, vol. 2773 de *Lecture Notes in Artificial Intelligence, subseries of LNCS*, páginas 906–913. Springer, 2003. ISBN 3-540-40803-7.

GÓMEZ MARTÍN, P. P., GÓMEZ MARTÍN, M. A. y GONZÁLEZ CALERO, P. A. Learning-by-doing through metaphorical simulation. En *9th International Conference, Knowledge-Based Intelligent Information and Engineering Systems, KES 2005* (editado por R. Khosla, R. J. Howlett y L. C. Jain), vol. 3682 de *Lecture Notes in Artificial Intelligence, subseries*

of LNCS, páginas 55–64. Springer, Melbourne, Australia, 2005d. ISBN 978-3-540-28895-4. ISSN 0302-9743.

GÓMEZ MARTÍN, P. P., GÓMEZ MARTÍN, M. A., GONZÁLEZ CALERO, P. A. y PALMIER CAMPOS, P. Using metaphors in game-based education. En *Technologies for E-Learning and Digital Entertainment. Second International Conference of E-Learning and Games (Edutainment'07)* (editado por K. chuen Hui, Z. Pan, R. C. kit Chung, C. C. Wang, X. Jin, S. Göbel y E. C.-L. Li), vol. 4469 de *Lecture Notes in Computer Science*, páginas 477–488. Springer-Verlag, 2007d. ISBN 3-540-73010-9.

GONZÁLEZ CALERO, P. A., GÓMEZ MARTÍN, P. P. y GÓMEZ MARTÍN, M. A. Nuevas tecnologías, nuevas oportunidades. *Comunicación y Pedagogía*, vol. 216, páginas 71–76, 2006. ISSN 1136-7733.

GOODMAN, B., SOLLER, A., LINTON, F. y GAIMARI, R. Encouraging student reflection and articulation using a learning companion. *International Journal of Artificial Intelligence in Education*, vol. 9, páginas 237–255, 1998.

GOSLING, J., JOY, B., STEELE, G. y BRACHA, G. *The Java Language Specification*. Prentice Hall, 3rd edición, 2005. ISBN 0-321-146780.

GRAEPEL, T., HERBRICH, R. y GOLD, J. Learning to fight. En *International Conference on Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004)* (editado por Q. Mehdi, N. Gough, S. Natkin y D. Al-Dabass), páginas 193–200. Microsoft Campus, 2004.

GRIEG, S., MUZYKA, R., OHLEN, J., OHLEN, T. y ZESCHUK, G. Postmortem: Bioware's neverwinter nights. 2002. Disponible en `http://www.gamasutra.com/features/20021204/grieg_pfv.htm` (último acceso, Agosto, 2007).

GYGAX, G. y ARNESON, D. *Dungeons & Dragons*. Tactical Studies Rules, 1974.

HAO, W. D. Postmortem: Tom Clancy's splinter cell. 2003. Disponible en `http://www.gamasutra.com/resource_guide/20030714/hao_pfv.htm` (último acceso, Agosto, 2007).

HAYES-ROTH, B. y DOYLE, P. Animate characters. *Autonomous Agents and Multi-Agent Systems*, vol. 1(2), páginas 175–204, 1998.

HEFFERNAN, N. T. *Intelligent Tutoring Systems have Forgotten the Tutor: Adding a Cognitive Model of Human Tutors*. Tesis Doctoral, School of Computer Science, Carnegie Mellon University, Pittsburgh, EEUU, 2001.

HIPP, J., GÜNTZER, U. y NAKHAEIZADEH, G. Algorithms for association rule mining - a general survey and comparison. *ACM SIGKDD Explorations Newsletter*, vol. 2(1), páginas 58–64, 2000. ISSN 1931-0145.

HOLLAN, J. D., HUTCHINS, E. y WEITZMAN, L. STEAMER: An interactive inspectable simulation-based training system. *AI Magazine*, vol. 5(2), páginas 15–27, 1984.

HOLLNAGEL, E. The phenotype of erroneous actions. *International Journal of Man-Machine Studies*, vol. 39(1), páginas 1–32, 1993. ISSN 0020-7373.

HOULETTE, R. *AI Game Programming Wisdom II*, capítulo Player Modeling for Adaptive Games, páginas 557–566. Charles River Media, 2004. ISBN 1-584-50289-4.

HOULETTE, R. y FU, D. *AI Game Programming Wisdom II*, capítulo The Ultimate Guide to FSMs in Games. Charles River Media, 2004. ISBN 1-584-50289-4.

HUEBNER, R. Using java as an embedded game scripting language. 1999.

HUEBNER, R. Postmortem of Nihilistic software's Vampire: The masquerade redemption. 2000. Disponible en `http://www.gamasutra.com/features/20000802/huebner_pfv.htm` (último acceso, Agosto, 2007).

IERUSALIMSCHY, R., DE FIGUEIREDO, L. H. y CELES, W. The implementation of lua 5.0. *Journal of Universal Computer Science*, vol. 11(7), páginas 1159–1176, 2005.

IERUSALIMSCHY, R., DE FIGUEIREDO, L. H. y CELES, W. *Lua 5.1 Reference Manual*. Lua.org, 2006. ISBN 8-590-37983-3.

IERUSALIMSCHY, R., DE FIGUEIREDO, L. H. y CELES, W. The evolution of Lua. En *The Third ACM SIGPLAN of Programming Languages Conference (HOPL-III)*. 2007.

INFOCOM. *Learning ZIL or Everything You Always Wanted to Know About Writing Interactive Fiction But Couldn't Find Anyone Still Working Here to Ask*. Infocom, Inc., 1989.

INOUE, Y. Methodological issues in the evaluation of intelligent tutoring systems. *Educational Technology Systems*, vol. 29(3), páginas 251–258, 2001.

JIMÉNEZ DÍAZ, G., GÓMEZ ALBARRÁN, M., GÓMEZ MARTÍN, M. A. y GONZÁLEZ CALERO, P. A. Software behaviour understanding supported by dynamic visualization and role-play. *SIGCSE Bulletin*, vol. 37(3), páginas 54–58, 2005a. ISSN 0097-8418.

JIMÉNEZ DÍAZ, G., GÓMEZ ALBARRÁN, M., GÓMEZ MARTÍN, M. A. y GONZÁLEZ CALERO, P. A. Understanding object-oriented software through virtual role-play. En *ICALT'05: Proceedings of the 5th IEEE International Conference on Advanced Learning Technologies*, páginas 875–877. IEEE Computer Society Press, Los Alamitos, California, USA, 2005b. ISBN 0-7695-2338-2.

JIMÉNEZ DÍAZ, G., GÓMEZ ALBARRÁN, M., GÓMEZ MARTÍN, M. A. y GONZÁLEZ CALERO, P. A. ViRPlay: Playing roles to understand dynamic behavior. En *9th Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts, at 19th European Conference on Object Oriented Programming (ECOOP)*. 2005c.

JOHNSON, W. L., RICKEL, J., STILES, R. y MUNRO, A. Integrating pedagogical agents into virtual environments. *Presence: Teleoperators and Virtual Environments*, vol. 7(6), páginas 523–546, 1998.

JOHNSON, W. L., RICKEL, J. W. y LESTER, J. C. Animated pedagogical agents: face-to-face interaction in interactive learning environments. *International Journal of Artificial Intelligence in Education*, vol. 11, páginas 47–78, 2000.

JOHNSON, W. L., VILHJALMSSON, H. y MARSELLA, S. Serious games for language learning: How much game, how much AI. En *12th International Conference on Artificial Intelligence in Education (AIED'05)*. IOS Press, 2005.

KAY, J. The UM toolkit for cooperative user models. *User Modeling and User-Adapted Interaction*, vol. 4(3), páginas 149–196, 1995.

KEARSLEY, G., editor. *Artificial Intelligence and Instruction. Applications and Methods*. Addison-Wesley, 1987.

KERIEVSKY, J. Stop over-engineering! *Software Development magazine*, 2002. Disponible en `http://www.ddj.com/dept/architect/184414835` (último acceso, Agosto, 2007).

KIM, J. H. Role–playing game page. 2006. Disponible en `http://www.darkshire.net/~jhkim/rpg/` (último acceso, Agosto, 2007).

KIRMSE, A., editor. *Game Programming Gems 4*. Charles River Media, 2004. ISBN 1-584-50295-9.

KOBSA, A. Supporting user interfaces for all through user modeling. En *Proceedings of the 6th International Conference on Human–Computer Interaction (HCI)*, páginas 155–157. 1995.

KODAGANALLUR, V., WEITZ, R. y ROSENTHAL, D. Tools for building intelligent tutoring systems. En *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06) Track 2*, vol. 2. IEEE Computer Society, Los Alamitos, CA, USA, 2006. ISSN 1530-1605.

KODICEK, D. *Mathematics and Physics for Programmers*. Charles River Media, 2005. ISBN 1-584-50330-0.

KOEDINGER, K. R., ALEVEN, V., HEFFERNAN, N., MCLAREN, B. y HOCKENBERRY, M. Opening the door to non-programmers: Authoring intelligent tutor behaviour by demonstration. En *Proceedings of the 7th International Conference on Intelligent Tutoring Systems (ITS'04)* (editado por J. C. Lester, R. M. Vicari y F. Paraguaçu), vol. 3220 de *Lecture Notes in Computer Science*, páginas 162–174. Springer-Verlag, 2004.

KOLODNER, J. L. Educational implications of analogy: A view from case-based reasoning. *American Psychologist*, vol. 52(1), páginas 57–66, 1997.

KUMAR, A. N. Model-based reasoning for domain modeling in a web-based intelligent tutoring system to help students learn to debug C++ programs. En *Proceedings of the 6th International Conference on Intelligent Tutoring Systems (ITS'02)* (editado por S. A. Cerri, G. Gouardères y F. Paraguaçu), vol. 2363 de *Lecture Notes in Computer Science*, páginas 792–801. Springer-Verlag, 2002.

KUZNETSOV, S. O. Machine learning and formal concept analysis. En *International Conference on Formal Concept Analysis (ICFCA'04)* (editado por P. Eklund), vol. 2961 de *Lecture Notes in Artificial Intelligence (subserie de LNCS)*, páginas 287–312. Springer-Verlag, 2004. ISBN 978-3-540-21043-6.

LAIRD, J. E., NEWELL, A. y ROSENBLOOM, P. S. SOAR: An architecture for general intelligence. *Artificial Intelligence*, vol. 33(1), páginas 1–64, 1987. ISSN 0004-3702.

LAKOS, J. *Large Scale C++ Software Design*. Addison Wesley, 1996. ISBN 0-201-63362-0.

LARSEN, S. Playing the game: Managing computer game development. 2002. Disponible en `http://www.blackwood.dk/PDF/PlayingTheGame-IE.pdf` (último acceso, Agosto, 2007).

LAWLER, R. W. y LAWLER, G. P. Computer microworlds and reading: an analysis for their systematic application. En *Artificial intelligence and education* (editado por R. W. Lawler y M. Yazdani), páginas 95–115. Ablex Publishing Corp., Norwood, NJ, USA, 1987.

LAWLER, R. W. y YAZDANI, M., editores. *Artificial Intelligence and Education*. Ablex Publishing, 1987.

LENGYEL, E. *The OpenAL Programming Guide*. Charles River Media, 2006. ISBN 1-584-503831.

LESTER, J. C., STONE, B. A. y STELLING, G. D. Lifelike pedagogical agents for mixed-initiative problem solving in constructivist learning environments. *User Modeling and User-Adapted Interaction*, vol. 9(1–2), páginas 1–44, 1999a. ISSN 0924-1868.

LESTER, J. C., ZETTLEMOYER, L. S., GRÉGOIRE, J. P. y BARES, W. H. Explanatory lifelike avatars: Performing user-centered tasks in 3D learning environments. En *Proceedings of the Third International Conference on Autonomous Agents*. ACM Press, 1999b.

LEUF, B. y CUNNINGHAM, W. *The Wiki Way: Quick Collaboration on the Web*. Addison-Wesley Professional, 2001.

LIANG, S. *Java(TM) Native Interface: Programmer's Guide and Specification*. Prentice Hall, 1999. ISBN 0-20132-577-2.

LINDHOLM, T. y YELLIN, F. *The Java Virtual Machine Specification*. Addison-Wesley Professional, 2nd edición, 1999.

LINDIG, C. y SNELTING, G. Assessing modular structure of legacy code on mathematical concept analysis. En *International Conference on Software Engineering (ICSE'97)*, páginas 349–359. IEEE Computer Society, 1997.

LIPPMAN, S. B. *Inside the C++ Object Model*. Addison Wesley, 1996. ISBN 0-201-83454-5.

LLOPIS, N. *Game Programming Gems 4*, capítulo The Clock: Keeping Your Finger on the Pulse of the Game. Charles River Media, 2004. ISBN 1-584-50295-9.

LLOPIS, N. *Introduction to Game Development*, capítulo Teams and Processes, páginas 163–182. Charles River Media, 2005a. ISBN 1-584-50377-7.

LLOPIS, N. *Introduction to Game Development*, capítulo Game Architecture, páginas 267–296. Charles River Media, 2005b. ISBN 1-584-50377-7.

LUCASARTS. Grim fandango. 1998.

LUEBKE, D. P. y GEORGES, C. Portals and mirrors: Simple, fast evaluation of potentially visible sets. En *Symposium on Interactive 3D Graphics*, páginas 105–106. ACM SIGGRAPH, 1995.

LUXENBURGUER, M. Implications partielles dans un contexte. *Mathématiques, Informatique et Sciences Humaines*, vol. 113(29), páginas 35–55, 1991.

MAES, P. Agents that reduce work and information overload. *Communications of the ACM*, vol. 37(7), páginas 30–40, 1994.

MARK, M. A. y GREER, J. E. Evaluation methodologies for intelligent tutoring systems. *Artificial Intelligence in Education*, vol. 4(2–3), páginas 129–153, 1993.

MARTEL, E. *AI Game Programming Wisdom III*, capítulo An Analysis of Far Cry: Instincts' Anchor System, páginas 555–566. Charles River Media, 2006. ISBN 1-584-50457-9.

MARTIN, K. y HOFFMAN, B. *Mastering CMake*. Kitware, Inc., 2006. ISBN 1-930-93416-5.

MARTÍNEZ-ORTÍZ, I., MORENO-GER, P., SIERRA, J. L. y FERNÁNDEZ-MANJÓN, B. Production and deployment of educational videogames as assessable learning objects. En *Innovative Approaches for Learning and Knowledge Sharing, First European Conference on Technology Enhanced Learning (EC-TEL 2006)* (editado por W. Nejdl y K. Tochtermann), vol. 4227 de *Lecture Notes in Computer Science*, páginas 316–330. Springer-Verlag, 2006.

MAUDUIT, C. U61. 2003. Disponible en `http://www.ufoot.org/u61` (último acceso, Agosto, 2007).

MCLEAN, A. *AI Game Programming Wisdom*, capítulo An Efficient AI Architecture using Prioritized Task Categories. Charles River Media, 2002. ISBN 1-584-50077-8.

MEYERS, S. *Effective C++: 50 Specific Ways to Improve Your Programs and Design*. Addison Wesley, 1997. ISBN 0-201-92488-9.

MOGILEFSKY, B. Lua in Grim fandango. 1999. Disponible en `http://www.grimfandango.net/?page=articles&pagenumber=2` (último acceso, Diciembre, 2006).

MOLLER, T., HAINES, E. y AKENINE-MOLLER, T. *Real-Time Rendering*. AK Peters, 2002. ISBN 1-568-81182-9.

MOLYNEUX, P. Postmortem: Lionhead studios'black & white. 2001. Disponible en `http://www.gamasutra.com/features/20010613/molyneux_pfv.htm` (último acceso, Agosto, 2007).

MÖNKKÖNEN, V. Multithreaded game engine architectures. 2006. Disponible en `http://www.gamasutra.com/features/20060906/monkkonen_01.shtml` (último acceso, Agosto, 2007).

MONSIEURS, P., CONINX, K. y FLERACKERS, E. Collision avoidance and map construction using synthetic vision. En *Workshop on Intelligent Virtual Agents (Virtual Agents'99)*. 1999.

MORENO-GER, P., MARTÍNEZ-ORTÍZ, I., SIERRA, J. L. y FERNÁNDEZ-MANJÓN, B. Language-driven development of videogames: The e-game experience. En *Fifth International Conference on Entertainment Computing (ICEC 2006)* (editado por R. Harper, M. Rauterberg y M. Combetto), vol. 4161 de *Lecture Notes in Computer Science*, páginas 153–164. Springer-Verlag, 2006. ISBN 3-540-45259-1.

MUNRO, A., JOHNSON, M. C., PIZZINI, Q. A., SURMON, D. S., TOWNE, D. M. y WOGULIS, J. L. Authoring simulation-centered tutors with RIDES. *International Journal of Artificial Intelligence in Education*, vol. 8, páginas 284–316, 1997.

MUNRO, A., SURMON, D. S., JOHNSON, M. C., PIZZINI, Q. A. y WALKER, J. P. An open architecture for simultaion-centered tutors. En *Proceedings of the 9th Conference on Artificial Intelligence in Education (AIED'99)*, páginas 360–367. 1999.

MURRAY, T. Authoring intelligent tutoring systems: An analysis of the state of art. *International Journal of Artificial Intelligence in Education*, vol. 10, páginas 98–129, 1999.

MUÑÓZ ÁVILA, H. y HOANG, H. *AI Game Programming Wisdom III*, capítulo Coordinating Teams of Bots with Hierarchical Task Network Planning, páginas 417–427. Charles River Media, 2006. ISBN 1-584-50457-9.

NELSON, G. *The Inform's Designer Manual*. The Interactive Fiction Library (IFLibrary.org), cuarta edición, 2001. ISBN 0-9713119-0-0.

NIHILISTIC-SOFTWARE. Vampire: The Masquerade Redemption. 2000.

NIJHOLT, A. y HEYLEN, D. Multimodal communication in inhabited virtual environments. *International Journal of Speech Technology*, vol. 5, páginas 343–353, 2002. ISSN 1381-2416.

OBITKO, M., SNASEL, V. y SMID, J. Ontology design with formal concept analysis. En *Proceedings of the International Workshop on Concept Lattices and their Applications (CLA'04)* (editado por V. Snasel y Belohlavek), páginas 111–119. VSB-Technical University of Ostrava, Dept. of Computer Science, 2004. ISBN 80-248-0597-9.

PALLISTER, K., editor. *Game Programming Gems 5*. Charles River Media, 2005. ISBN 1-584-50352-1.

PANAYIOTOPOULOS, T., ZACHARIS, N. y VOSINAKIS, S. Intelligent guidance in a virtual university. En *IMACS/IFAC International Symposium on Soft Computing in Engineering Applications, SOFTCOM '98*. 1998.

PAPATHEODOROU, C. Machine learning in user modeling. En *Machine Learning and Its Applications* (editado por G. Paliouras, V. Karkaletsis y C. D. Spyropoulos), vol. 2049 de *Lecture Notes in Computer Science*, páginas 286–294. Springer-Verlag, 2001. ISBN 3-540-42490-3.

PASQUIER, N. Mining association rules using formal concept analysis. En *International Conference on Conceptual Structures (ICCS'2000)*, páginas 259–264. Shaker Verlag, 2000.

PEINADO GIL, F., GÓMEZ MARTÍN, P. P. y GÓMEZ MARTÍN, M. A. A game architecture for emergent story-puzzles in a persistent world. En *DIGRA 2005 - Changing Views: Worlds in Play - Electronic Proceedings*. 2005.

PIAGET, J. *The Construction of Reality in the Child*. New York: Basic Books, 1955.

PLUMMER, J. *A Flexible and Expandable Architecture for Computer Games*. Proyecto Fin de Carrera, Arizona State University, 2004.

POLSON, M. y RICHARDSON, J., editores. *Foundations of Intelligent Tutoring Systems*. Lawrence Erlbaum Associates, 1988.

PONSEN, M. y SPRONCK, P. Improving adaptive game ai with evolutionary learning. En *Computer Games: Artificial Intelligence, Design and Education (CGAIDE'04)* (editado por Q. Mehdi, N. Gough, S. Natkin y D. Al-Dabass), páginas 389–396. 2004.

PRITCHAR, M. *Game Programming Gems 2*, capítulo A High-Performance Tile-based Line-of-sight and Search System, páginas 279–286. Charles River Media, 2001. ISBN 1-584-50054-9.

RAVIN, S., editor. *AI Game Programming Wisdom*. Charles River Media, 2002. ISBN 1-584-50077-8.

RAVIN, S., editor. *AI Game Programming Wisdom II*. Charles River Media, 2004. ISBN 1-584-50289-4.

RAVIN, S., editor. *AI Game Programming Wisdom III*. Charles River Media, 2006. ISBN 1-584-50457-9.

RECIO, J. A., DÍAZ AGUDO, B., GÓMEZ MARTÍN, M. A. y WIRATUNGA, N. Extending jCOLIBRI for textual CBR. En *Proceedings of Case-Based Reasoning Research and Development, 6th International Conference on Case-Based Reasoning, ICCBR 2005* (editado por H. Muñoz-Avila y F. Ricci), vol. 3620 de *Lecture Notes in Artificial Intelligence, subseries of LNCS*, páginas 421–435. Springer, Chicago, IL, US, 2005. ISBN 978-3-540-28174-0. ISSN 0302-9743.

RECIO-GARCÍA, J. A., SÁNCHEZ-RUIZ, A. A., DÍAZ-AGUDO, B. y GONZÁLEZ-CALERO, P. A. jCOLIBRI 1.0 in a nutshell. a software tool for designing cbr systems. En *Proccedings of the 10th UK Workshop on Case Based Reasoning* (editado por M. Petridis), páginas 20–28. CMS Press, University of Greenwich, 2005.

REINHART, B. Postmortem: Epic games' Unreal Tournament. 2000. Disponible en `http://www.gamasutra.com/features/20000609/reinhart_pfv.htm` (último acceso, Agosto, 2007).

REYES, R. L. y SISON, R. C. CBR-Tutor: A case-based reasoning approach to an internet agent-based tutoring system. En *International Conference on Computers in Education (ICCE'00)*, vol. 2, páginas 1413–1420. 2000.

REYNOLDS, C. Steering behaviors for autonomous characters. En *Game Developers Conference*, páginas 763–782. 1999.

RHALIBI, A. E., MERABTI, M. y SHEN, Y. Improving game processing in multithreading and multiprocessor architecture. En *Proceedings of the I International Conference on Technologies for E-Learning and Digital Entertainment (Edutainment'06)* (editado por Z. Pan, R. Aylett, H. Diener, X. Jin, S. Göbel y L. Li), vol. 3942 de *Lecture Notes in Computer Science*, páginas 669–679. Springer-Verlag, 2006.

RICHTER, J. *Programación avanzada en Windows*. Microsoft Press, 1997. ISBN 84-481-1160-5.

RICKEL, J. Intelligent virtual agents for education and training: Opportunities and challenges. En *Proceedings of the Third International Workshop Intelligent Virtual Agents (IVA'01)* (editado por A. de Antonio, R. Aylett y D. Ballin), vol. 2190 de *Lecture Notes in Computer Science*, páginas 15–22. Springer-Verlag, 2001. ISBN 3-540-42570-5.

RICKEL, J. y JOHNSON, W. L. Intelligent tutoring in virtual reality: A preliminary report. En *Proceedings of the Eighth World Conference on Artificial Intelligence in Education*, páginas 294–301. IOS Press., 1997.

RICKEL, J. y JOHNSON, W. L. STEVE: A pedagogical agent for virtual reality. En *2ª International Conference on Autonomous Agents* (editado

por K. P. Sycara y M. Wooldridge), páginas 332–333. ACM Press, New York, 1998. ISBN 0-89791-983-1.

RICKEL, J. y JOHNSON, W. L. Animated agents for procedural training in virtual reality: Perception, cognition, and motor control. *Applied Artificial Intelligence*, vol. 13, páginas 343–382, 1999.

RICKEL, J. y JOHNSON, W. L. Task-oriented collaboration with embodied agents in virtual worlds. En *Embodied conversational agents* (editado por J. Cassell, J. Sullivan y S. Prevost), páginas 95–122. MIT Press, Cambridge, MA, USA, 2000. ISBN 0-262-03278-3.

ROLLINGS, A. y ADAMS, E. *Andrew Rollings and Ernest Adams on Game Design*. New Riders Games, 2003. ISBN 1-592-73001-9.

ROLLINGS, A. y MORRIS, D. *Game Architecture and Design*. Coriolis Group, 1999.

ROLLINGS, A. y MORRIS, D. *Game Architecture and Design: A new edition*. New Riders Publishing, segunda edición edición, 2004.

ROSADO, G. *AI Game Programming Wisdom II*, capítulo Implementing a Data-Driven Finite-State Machine. Charles River Media, 2004. ISBN 1-584-50289-4.

VAN ROSSUM, G. *Python Reference Manual, Release 2.5*. Python Software Foundation, 2006.

ROZANSKI, N. y WOODS, E. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison Wesley Professional, 2005.

RUCKER, R. *Software Engineering and Computer Games*. Addison Wesley, 2002. ISBN 0-201-76791-0.

RUSSELL, S. J. y NORVIG, P. *Artificial Intelligence: A Modern Approach*. Prentice Hall, primera edición, 1995. ISBN 0-13-103805-2.

SANCHEZ-CRESPO DALMAU, D. *Core Techniques and Algorithms in Game Programming*. New Riders Games, 2003. ISBN 0-131-02009-9.

SÁNCHEZ PELEGRÍN, R., GÓMEZ MARTÍN, M. A. y DÍAZ AGUDO, B. A CBR module for a strategy videogame. En *1st Workshop on Computer Gaming and Simulation Environments, at 6th International Conference on Case-Based Reasoning (ICCBR)* (editado por D. W. Aha y D. Wilson), páginas 217–226. Chicago, IL, US, 2005.

SANSONE, C. y HARACKIEWICZ, J. M. *Intrinsic and Extrinsic Motivation: The Search for Optimal Motivation and Performance*. Academic Press, 2000. ISBN 0-12-619070-4.

DOS SANTOS, C. T. y OSÓRIO, F. S. AdapTIVE: An intelligent virtual environment and its application in e-commerce. En *28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, vol. 1, páginas 468–473. IEEE Computer Society, Los Alamitos, CA, USA, 2004a. ISSN 0730-3157.

DOS SANTOS, C. T. y OSÓRIO, F. S. Integrating intelligent agents, user models, and automatic content categorization in a virtual environment. En *Proceedings of the 7th International Conference on Intelligent Tutoring Systems (ITS'04)* (editado por J. C. Lester, R. M. Vicari y F. Paraguaçu), vol. 3220 de *Lecture Notes in Computer Science*, páginas 128–139. Springer-Verlag, 2004b.

DOS SANTOS, C. T. y OSÓRIO, F. S. An intelligent and adaptive virtual environment and its application in distance learning. En *AVI '04: Proceedings of the working conference on Advanced Visual Interfaces*, páginas 362–365. ACM Press, New York, NY, USA, 2004c. ISBN 1-58113-867-9.

SCHANK, R. y CLEARYV, C. *Engines for Education*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1994.

SCHWAB, B. *AI Game Engine Programming*. Charles River Media, 2004. ISBN 1-584-50344-0.

SHAW, E., JOHNSON, W. L. y GANESHAN, R. Pedagogical agents on the web. En *Proceedings of the third annual conference on Autonomous Agents*, páginas 283–290. ACM Press, 1999. ISBN 1-58113-066-X.

SHAW, M. y GARLAN, D. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

DA SILVA, F. S. C. y VASCONCELOS, W. W. Rule schemata for game artificial intelligence. En *Advances in Artificial Intelligence, proceedings of the International Joint Conference IBERAMIA-SBIA*, vol. 4140 de *Lecture Notes in Computer Science*, páginas 451–461. Springer-Verlag, 2006. ISBN 978-3-540-45462-5.

SLEEMAN, D. H. y BROWN, J. S., editores. *Intelligent Tutoring Systems*. Academic Press, London, 1982.

SLOAN, J. y MULL, W. Doom 3 faq. 2003. Disponible en `http://www.newdoom.com/newdoomfaq.php` (último acceso, Agosto, 2007).

SNELTING, G. Concept lattices in software analysis. En *Formal Concept Analysis, Foundations and Applications* (editado por B. Ganter, G. Stumme y R. Wille), vol. 3626 de *Lecture Notes in Computer Science*, páginas 272–297. Springer-Verlag, 2005. ISBN 3-540-27891-5.

SPRONCK, P., PONSEN, M., SPRINKHUIZEN-KUYPER, I. y POSTMA, E. Adaptive game ai with dynamic scripting. *Machine Learning*, vol. 63(3), páginas 217–248, 2006.

SPRONCK, P. H. M. *Adaptive Game AI*. Tesis Doctoral, Dutch Research School for Information and Knowledge Systems, 2005.

STONE, B. A. y LESTER, J. C. Dynamically sequencing an animated pedagogical agent. En *Readings in agents*, páginas 156–163. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. ISBN 1-55860-495-2.

STOTTLER, R. H. y VINKAVICH, M. Tactical action officer intelligent tutoring system (TAO ITS). En *Proceedings of the Industry/Interservice, Training, Simulation & Education Conference (I/ITSEC 2000)*. 2000.

STUMME, G. Conceptual knowledge discovery with frequent concept lattices. Informe Técnico FB4-Preprint 2043, TU Darmstadt, 1999.

SWEENEY, T., HENDRIKS, M. y PRESTENBACK, R. Unrealscript language reference. 2004. Disponible en `http://udn.epicgames.com/Two/UnrealScriptReference` (último acceso, Agosto, 2007).

SWENG-GAMEDEV. Software engineering, as it applies to game development. 2002. Disponible en `http://lists.midnightryder.com/listinfo.cgi/sweng-gamedev-midnightryder.com` (último acceso, Agosto, 2007).

TENNYSON, R., CHRISTENSEN, D. y PARK, S. The Minnesota adaptive instructional system: An intelligent CBI system. *Journal of Computer-Based Instruction*, vol. 11(1), páginas 2–13, 1984.

THALMANN, D. The foundations to build a virtual human society. En *Proceedings of the Third International Workshop Intelligent Virtual Agents (IVA'01)* (editado por A. de Antonio, R. Aylett y D. Ballin), vol. 2190 de *Lecture Notes in Computer Science*, páginas 1–14. Springer-Verlag, 2001. ISBN 3-540-42570-5.

THE ELECTRONIC ENTERTAINMENT EXPO. Exhibitor list. 2006. Disponible en `http://www.e3expo.com/exhibitors/exhibitor-list.aspx` (último acceso, Mayo, 2006).

THOMPSON, P. W. Mathematical microworlds and intelligent computer-assisted instruction. En *Artificial Intelligence and Instruction: Applications and Methods* (editado por G. Kearsley), páginas 83–109. Addison-Wesley, Boston, MA, USA, 1987.

TILLEY, T. Tool support for FCA. En *International Conference on Formal Concept Analysis (ICFCA'04)* (editado por P. Eklund), vol. 2961 de *Lecture Notes in Artificial Intelligence (subserie de LNCS)*, páginas 104–111. Springer-Verlag, 2004. ISBN 978-3-540-21043-6.

TRACZ, W. *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*. Addison Wesley, 1995a. ISBN 0201633698.

TRACZ, W. Confessions of a used-program salesman: lessons learned. En *Symposium on Software Reusability archive, Proceedings of the 1995 Symposium on Software reusability*, páginas 11–13. ACM Press, New York, EEUU, 1995b. ISBN 0-89791-739-1.

TREGLIA, D., editor. *Game Programming Gems 3*. Charles River Media, 2002. ISBN 1-584-50233-9.

TURNER, R. Saints Row multiprocessing architecture. En *Proceedings of the Game Developers Conference (GDC'07)*. 2007.

UHR, L. Teaching machine programs that generate problems as a function of interaction with students. En *Proceedings of the 24th National Conference*, páginas 125–134. ACM Press, New York, NY, USA, 1969.

VALENTE, L., CONCI, A. y FEIJÓ, B. Real time game loop models for single-player computer games. En *Simpósio Brasileiro de Jogos para Computador e Entretenimento Digital (SBGAMES)*. 2005.

VALVE SOFTWARE. Half life. 1998.

VARANESE, A. *Game Scripting Mastery*. Course Technology PTR, 2002. ISBN 1-931-84157-8.

VESPERMAN, J. *Essential CVS*. O'Reilly, 2003. ISBN 0-596-00459-1.

VOLITION INC. Saints Row. 2006.

VOSINAKIS, S., ANASTASSAKIS, G. y PANAYIOTOPOULOS, T. DIVA: Distributed intelligent virtual agents. En *Proceedings of the Second International Workshopf on Intelligent Virtual Agents (IVA'99)*. 1999. ISSN 1467-2154.

VOSINAKIS, S. y PANAYIOTOPOULOS, T. Design and implementation of synthetic humans for virtual environments and simulation systems. En *Advances in Signal Processing and Computer Technologies* (editado por G. Antoniou, N. Mastorakis y O. Planfilov), Electrical and Computer Engineering, páginas 315–320. WSES Press, 2001a.

VOSINAKIS, S. y PANAYIOTOPOULOS, T. SimHuman: A platform for real-time virtual agents with planning capabilities. En *Proceedings of the Third*

International Workshop on Intelligent Virtual Agents (IVA'01) (editado por A. de Antonio, R. Aylett y D. Ballin), vol. 2190 de *Lecture Notes in Artificial Intelligence (subserie de LNCS)*, páginas 210–223. Springer-Verlag, 2001b.

WACHSMUTH, I. y KOPP, S. Lifelike gesture synthesis and timing for conversational agents. En *GW '01: Revised Papers from the International Gesture Workshop on Gesture and Sign Languages in Human-Computer Interaction*, páginas 120–133. Springer-Verlag, London, UK, 2002. ISBN 3-540-43678-2.

WARDELL, B. From the trenches. 1998. Disponible en `http://www.stardock.com/stardock/articles/Trenches/Trenches1198.html` (último acceso, Agosto, 2007).

WATT, A. H. *3D Computer Graphics*. Addison Wesley, tercera edición, 1999. ISBN 0-201-39855-9.

WENGER, E. *Artificial Intelligence and Tutoring Systems: Computational and Cognitive Approaches to the Communication of Knowledge*. Morgan Kaufmann, 1987.

WEST, M. Evolve your hiearchy. *Game Developer*, vol. 13(3), páginas 51–54, 2006.

WIKIPEDIA (PLATO). Entrada: "Plato system". Disponible en `http://en.wikipedia.org/wiki/PLATO_System` (último acceso, Agosto, 2007).

WILLE, R. *Restructuring lattice theory: an approach based on hierarchies of concepts*. Ordered Sets, 1982.

WILLE, R. Knowledge acquisition by methods of formal concept analysis. En *Proceedings of the conference on Data analysis, learning symbolic and numeric knowledge* (editado por E. Diday), páginas 365–380. Nova Science Publishers, 1989.

WILLE, R. Concept lattices and conceptual knowledge systems. *Computers and mathematics with applications*, vol. 23(6–9), páginas 493–515, 1992. ISSN 0898-1221.

WOLFF, K. E. A first course in formal concept analysis: How to understand line diagrams. *Advances in Statistical Software*, páginas 429–438, 1993.

WOOLDRIDGE, M. Intelligent agents. En *Multiagents Systems. A Modern Approach to Distributed Artificial Intelligence* (editado por G. Weiss), páginas 27–77. MIT Press, 1999.

YANNAKAKIS, G. N. y MARAGOUDAKIS, M. Player modeling impact on player's entertainment in computer games. En *Proceedings of the 10th International Conference on User Modeling* (editado por L. Ardissono, P. Brna y A. Mitrovic), vol. 3538 de *Lecture Notes in Artificial Intelligence (subserie de LNCS)*, páginas 74–78. Springer-Verlag, 2005. ISBN 3-540-27885-0.

Lista de acrónimos

AI ..... *Artificial Intelligence*, Inteligencia Artificial

BSP ..... *Binary Space Partitioning*, Particionado binario del espacio

CAI ..... *Computer-Aided Instruction*, Enseñanza asistida por ordenador

CBR ..... *Case-based reasoning*, Razonamiento basado en casos

COTS ..... *Component off the shelf*

CVS ..... *Concurrent Versions System*

DLL ..... *Dynamic Linking Library*, Biblioteca de Enlace Dinámico

E3 ..... *The Electronic Entertainment Expo*

FAQ ..... *Frecuently asked questions*

FCA ..... *Formal Concept Analysis*, Análisis formal de conceptos

FPS ..... *First Person Shooters*, Acción en primera persona

FPS ..... Fotogramas por segundo

GPA ..... *Generalized Pedagogical Agent*

GPL ..... *General Public License*

GrimE ..... *Grim Edit*

IA ..... Intelingecia Artificial

ICAI ..... *Intelligent Computer-Aided Instruction*

IDE ..... *Integrated development environment*, Entorno integrado de desarrollo

ITS ..... *Intelligent Tutoring System*, Sistema Inteligente de Tutoría

IVA ..... *Intelligent Virtual Agent*, Agente virtual inteligente

IVE ..... *Intelligent Virtual Environment*, Entorno virtual inteligente

JIT ..... *Just-in time code generation*

JNI ..... *Java Native Interface*

JVM ..... *Java Virtual Machine*, Máquina virtual de Java

KMS ..... *Knowledge management system*, Sistema de gestión del conocimiento

LGPL ..... *Lesser General Public License*

LMS ..... *Learning Management System*, Sistema gestor de aprendizaje

NPC ..... *Non-Player Character*, Personaje no jugador

NTSC ..... *National Television System Commitee*

OIM ..... *Object Interface Manager*

PAL ..... *Phase Alternating Line*, Línea alternada en fase

PLATO ..... *Programmed Logic Automated Teaching Operations*

QA ..... *Quality assurance*

SCORM ..... *Sharable Content Object Reference Model*

SCUMM ..... *Script Creation Utility for Maniac Mansion*, Utilidad de creación de guiones para Maniac Mansion

SDK ..... *Software Development Kit*, Kit de desarrollo software

SMP ..... *Symmetric Multi-Processing*, Multiproceso simétrico

XML ..... *eXtensible Markup Language*, Lenguaje extensible de marcado

ZIL ..... *Zork Implementation Language*, Lenguaje de implementación de Zork

–¿Qué te parece desto, Sancho? – Dijo Don Quijote –
Bien podrán los encantadores quitarme la ventura,
pero el esfuerzo y el ánimo, será imposible.

Segunda parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes

–Buena está – dijo Sancho –; fírmela vuestra merced.
–No es menester firmarla – dijo Don Quijote–,
sino solamente poner mi rúbrica.

Primera parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes