



UNIVERSIDADE DA CORUÑA

FACULTAD DE INFORMÁTICA
DEPARTAMENTO DE COMPUTACIÓN

PROYECTO DE FIN DE CARRERA
DE INGENIERÍA INFORMÁTICA

Diseño e implementación del juego en línea multijugador masivo “Thearsmonsters” basado en tecnologías Web 2.0.

Autor: Mario Izquierdo Martínez
Director: Víctor M. Gulías Fernández

A Coruña, a 21 de octubre de 2009.

Información general

Título del proyecto: “Concepto, Análisis, Diseño e Implementación de un juego en línea multijugador masivo basado en navegador: Thearsmonsters”

Clase de proyecto: Proyecto Clásico de Ingeniería

Nombre del alumno: Mario Izquierdo Martínez

Nombre del director: Víctor Manuel Gulías Fernández

Miembros del tribunal:

Miembros suplentes:

Fecha de lectura:

Calificación:

D. Víctor M. Gulías Fernández

CERTIFICA

Que la memoria titulada “**Concepto, Análisis, Diseño e Implementación de un juego en línea multijugador masivo basado en navegador: Thearsmonsters**” ha sido realizada por Mario Izquierdo Martínez con D.N.I. 36.123.605-N bajo la dirección de D. Víctor M. Gulías Fernández. La presente constituye la documentación que, con mi autorización, entrega el mencionado alumno para optar a la titulación de Ingeniero en Informática.

A Coruña, a 21 de octubre de 2009.

Este proyecto va dedicado a mi difunta abuela Flora María Oliveira Requejo, que siempre tenía una sonrisa para sus nietos. Ella, aun sin saber muy bien para qué servía, me compró mi primer ordenador cuando tenía solo 10 años, el cual ha sido una pieza clave para definir mi vocación por la informática y en especial el diseño de videojuegos.

Agradecimientos

La parte artística de este proyecto ha sido realizado con la colaboración de *Ángel Romo Sandoval*, estudiante de Arquitectura, pintor artístico y excelente diseñador gráfico. Sin su ayuda y motivación el juego “Thearsmonsters” no sería lo mismo.

Desde luego no se habría alcanzado la calidad obtenida si no fuese por el asesoramiento de *Víctor M. Gulías*, director del proyecto, que además me ha facilitado una plaza en su laboratorio donde está todo el material necesario para el desarrollo del proyecto, y tambien están *Javier París* y , que solucionaron todos los problemas tecnológicos y de configuración que han aparecido. Su constante atención y singular ayuda sobre todo en la parte técnica hicieron posible alcanzar los objetivos.

Mención especial a mis padres, que siempre me apoyaron con todo lo necesario para poder llegar hasta aquí y estoy seguro de que lo seguirán haciendo en el futuro.

Gracias también a la empresa *Interacción* por apoyar todas mis ideas y enseñarme lo que es el desarrollo Web en la vida real. Pienso que las prácticas realizadas allí durante el verano del 2007 no me han hecho perder las vacaciones, y además me han ayudado a conocer el trabajo que realmente me gusta.

Así mismo gracias a todas aquellas personas que escucharon muchas de las ideas que propone el juego y procuraron dar una opinión objetiva, como mis hermanos *Francisco* y *Adriana*, mis amigos *Santi, Alex, Manu, Pablo, Castro, Henrique, Iago* y muchos más que también han colaborado.

También es notable el esfuerzo que hizo mi novia *Jennifer* al esperar pacientemente y comprender todas las horas dedicadas al proyecto, además de no dudar a la hora de ofrecer su ayuda incondicional.

A todos muchas gracias.

Resumen

Este proyecto abarca el proceso completo (concepto, análisis, diseño e implementación) para crear de una aplicación Web destinada al entretenimiento interactivo en Internet. Se pretende concebir la franquicia *Thearsmonsters* desde sus cimientos hasta el despliegue final, teniendo en cuenta los aspectos de ingeniería del software necesarios para crear un producto de calidad y sostenible, requisito fundamental en una aplicación de estas características.

El proceso comienza por el análisis de las oportunidades de negocio existentes en el sector del videojuego, industria que crece de forma exponencial, donde el dinamismo que la caracteriza crea oportunidades de negocio de forma constante.

El análisis conceptual y el guión original del juego también forman parte del proyecto, así como el proceso creativo y otros criterios primordiales del diseño de videojuegos en general. Uno de estos principios es fomentar la participación y competencia entre jugadores.

Otra importante tarea es la elaboración del diseño gráfico de la interfaz de usuario, que es una parte elemental en el sector del entretenimiento. Una vez realizado el diseño artístico y fijados los elementos estéticos principales, hay que maquetar y dibujar cada uno de los componentes de la interfaz: personajes, escenarios, ventanas, botones, menús, iconos, etc.

Otro aspecto interesante es el análisis completo del sistema (persistencia, usuarios, conceptos modelados, etc.), se diseña el sistema de información usando el lenguaje de modelado UML y se elige la tecnología más adecuada. A nivel técnico, se representa el dominio en objetos, se aplican patrones conocidos y se prepara el modelo de forma que quede lo más adaptable y abierto para mantenimiento correctivo y evolutivo.

Finalmente, se implementa de un amplio subconjunto del modelo. Para ello se utiliza, entre otras tecnologías, el *framework Struts*, que funciona sobre la plataforma *Java EE* y JSP.

Palabras clave:

- ✓ Aplicación Web
- ✓ Videojuegos
- ✓ Java EE
- ✓ Struts
- ✓ Ajax
- ✓ Diseño gráfico
- ✓ Interfaces de usuario
- ✓ Patrones de diseño

Índice general

	Página
1. Introducción	1
1.1. Visión general	1
1.2. Objetivos del proyecto	2
1.3. Fases del proyecto	3
1.4. Descripción del contenido de la memoria	5
2. Contextualización	7
2.1. Desarrollo web	8
2.2. Diseño de videojuegos	9
2.3. Juegos MMOG	11
2.4. Juegos <i>MMORPG</i> basados en navegador	15
2.5. Análisis de mercado	17
2.6. Juegos de referencia	19
2.6.1. Ogame	20
2.6.2. Travian	25
2.6.3. World Of Warcraft	28
3. Concepto y Análisis	35
3.1. Resumen del juego	36
3.2. Sistema de juego	39
3.2.1. Tareas programables	39
3.2.2. Economía	41

3.2.3.	Misiones	43
3.2.4.	Ciclo generacional	44
3.3.	Modelo conceptual	45
3.4.	Usuarios	46
3.4.1.	Jugadores	46
3.4.1.1.	Perfil de jugador	47
3.4.1.2.	Organización de los jugadores: Alianzas y Amigos	48
3.4.1.3.	Comunicación entre jugadores: Mensajes	49
3.4.1.4.	Competencia entre jugadores	50
3.4.2.	Administradores	51
3.5.	Salas de la guarida	52
3.5.1.	Como se usan	53
3.5.2.	Realizar obras en una sala	54
3.5.2.1.	Crear salas nuevas	56
3.5.2.2.	Ampliar	56
3.5.2.3.	Actualizar	56
3.5.3.	Publicación	57
3.5.4.	Restricciones de acceso	58
3.5.5.	Tipos de sala disponibles	59
3.6.	Monstruos	61
3.6.1.	Estado de un monstruo	61
3.6.1.1.	Características genéticas raciales	62
3.6.1.2.	Atributos simples	64
3.6.1.3.	Atributos compuestos	65
3.6.1.4.	Habilidades aprendidas	66
3.6.1.5.	Condicionantes	66
3.6.2.	Ciclo de vida de los monstruos	67
3.6.3.	Monstruos en acción	69
3.7.	Misiones	70
3.7.1.	Realización de asaltos en una misión	70
3.7.2.	Templos y salas con PNJs	71
3.7.3.	Sistema para desbloquear razas y salas	72

3.8.	Batallas	73
3.8.1.	Influencia de las habilidades de batalla	74
3.8.2.	Inicialización de la batalla	74
3.8.3.	Inicialización de las rondas	74
3.8.4.	Factores de cada turno	76
3.8.5.	Finalización de la batalla	76
4.	Diseño e Implementación	77
4.1.	Subconjunto del análisis seleccionado para la primera versión	79
4.2.	Modelo y persistencia de datos	80
4.2.1.	Estructura de la base de datos	81
4.2.2.	Acceso a la base de datos	84
4.2.3.	Fachadas del modelo	87
4.3.	Conceptos modelados	90
4.3.1.	Usuarios	92
4.3.2.	Guaridas	94
4.3.3.	Salas	96
4.3.4.	Monstruos	98
4.3.5.	Tareas	103
4.3.6.	Configuración	105
4.4.	Controlador y Vista	107
4.4.1.	Estructura condicionada a JSTL y <i>Apache Struts</i>	108
4.4.2.	SessionManager	110
4.4.3.	Acciones por defecto	113
4.4.4.	Filtros Incluidos	115
4.4.5.	<i>Custom Tags</i> para simplificar la vista	117
4.5.	Internacionalización	122
4.5.1.	Internacionalización de aplicaciones con <i>Struts</i>	123
4.5.2.	Internacionalización y Localización con JSTL	123
4.6.	Vistas interactivas con <i>JavaScript</i>	127
4.6.1.	Importancia de la librería <i>JQuery</i>	128
4.6.2.	Uso de <i>JavaScript</i> en <i>Thearsmonsters</i>	129
4.7.	Acciones <i>Ajax</i>	131

4.7.1.	Elección del método para implementar <i>Ajax</i> en <i>Struts</i>	131
4.7.2.	Ejemplo de implementación con el método <i>Ajax</i> seleccionado . .	133
4.7.2.1.	Código <i>JavaScript</i>	135
4.7.2.2.	Configuración en <i>Struts-config</i> y <i>tiles-def</i>	136
4.7.2.3.	Acciones Struts	136
4.7.2.4.	Páginas JSP de la vista	138
4.7.2.5.	Añadir más acciones <i>Ajax</i> sobre esta estructura	141
5. Interfaz Gráfica		143
5.1.	Breve introducción al diseño Web	144
5.2.	Imagen digital	147
5.3.	Arte gráfica en <i>Thearsmonsters</i>	152
5.3.1.	Portada del juego	153
5.3.2.	Dentro del juego	155
5.3.3.	Representación gráfica de la guarida	160
5.3.4.	Representación gráfica de los monstruos	164
5.3.4.1.	Razas de monstruo	167
6. Conclusiones		171
6.1.	Resultados obtenidos	171
6.2.	Trabajo Futuro	173
A. Elección del <i>framework</i> Web		177
B. Breve introducción a Ajax		183
C. Juegos MMORPG para navegador de referencia		187
D. Manual para conectarse y utilizar la aplicación		189
E. Contenidos del CD		191
F. Glosario de acrónimos		193
G. Glosario de términos		195

Índice de figuras

Figura	Página
1.1. Logotipo del juego <i>Thearsmonsters</i>	2
1.2. Esquema de las fases por las que pasa el proyecto	3
2.1. Ejemplo de MMORPG tradicional: los jugadores se agrupan para charlar en una plaza del juego <i>Guildwars</i>	11
2.2. Ejemplo de MMORPG para navegador: alineación de un equipo en <i>Hattrick</i> . El juego se visualiza y controla desde un navegador Web como por ejemplo Firefox.	13
2.3. <i>Ogame</i> , juego espacial basado en navegador	21
2.4. Captura de pantalla de <i>Ogame</i> : Edificios de un planeta.	22
2.5. <i>Travian</i> , juego de batallas medievales para navegador	26
2.6. Representación de las aldeas en <i>Travian</i> . El crecimiento de la aldea es tangible.	27
2.7. Logotipo del WoW	29
2.8. Algunos personajes del <i>WoW</i>	30
2.9. Captura de pantalla en uno de los múltiples escenarios del <i>WoW</i>	32
3.1. Esquema básico del sistema de juego	37
3.2. El usuario asigna tareas a un monstruo y el sistema las realiza de manera automática.	40
3.3. Flujo de los recursos económicos. Dónde se obtienen, almacenan y utilizan en el juego.	42

3.4.	Los huevos pueden costar dinero (dependiendo de la raza de monstruo son más caros o más baratos), o también pueden suponer una fuente de ingresos, si el jugador cuenta con los medios necesarios para producirlos.	44
3.5.	Una primera aproximación al modelo conceptual del dominio.	45
3.6.	Usuarios del juego y su organización	47
3.7.	Casos de Uso de un miembro de una alianza. Algunos de ellos solo se pueden realizar si se dispone de los privilegios adecuados.	49
3.8.	Los mensajes que puede enviar o recibir un jugador también dependen de los privilegios dentro de su alianza.	50
3.9.	Casos de uso diferenciando a los jugadores normales de los administradores.	51
3.10.	Las acciones que son exclusivas de un administrador son comprobadas por el sistema.	52
3.11.	Estructura de las salas de la guarida.	53
3.12.	Casos de uso del jugador relacionados con la guarida.	54
3.13.	Diagrama de estados de una sala de la guarida (puede variar ligeramente dependiendo del tipo concreto de sala, pero casi siempre se mantiene este esquema).	55
3.14.	Los monstruos de una guarida pueden utilizar todas las salas de la misma. En otras guaridas solamente podrán acceder a las salas públicas, pagándole al dueño el precio establecido por el servicio. Así aparece un nuevo mercado dentro del juego.	58
3.15.	Diagrama de clases con la estructura de los monstruos del juego, sin entrar en detalles.	62
3.16.	Diferentes estados por los que pasa la vida de un monstruo.	68
3.17.	Diagrama de clases con la estructura de los monstruos del juego, sin entrar en detalles.	69
3.18.	Proceso mediante el cual un jugador selecciona un grupo de monstruos para realizar un nuevo asalto a un templo.	72
3.19.	Realización de una batalla entre dos grupos de monstruos.	75
4.1.	Conceptos para implementar en la primera versión del juego.	80
4.2.	Estructura de la base datos relacional que da soporte a la persistencia de datos en <i>Thearsmonsters</i>	81

4.3. Estructura general que siguen los DAOs implementados en la aplicación.	85
4.4. Driver JDBC. El programador siempre trabaja contra los paquetes <i>java.sql</i> y <i>javax.sql</i> , que forman parte de <i>Java SE</i> .	86
4.5. Estructura genérica de las fachadas del modelo junto con sus <i>PlainActions</i> . Las clases del paquete <i>j2ee.util.sql</i> son comunes a todas las fachadas.	88
4.6. Ejemplo de ejecución de la acción <code>register_user</code> en la fachada de usuarios.	89
4.7. Implementación de la capa modelo aplicando a su vez el patrón arquitectónico <i>Layers</i> .	90
4.8. Estructura genérica en capas que sigue cada uno de los componentes del modelo.	91
4.9. Clases del modelo que se encargan de gestionar las cuentas de usuario.	93
4.10. Clases del modelo relacionadas con la guarida del jugador.	95
4.11. Diagrama de clases de las salas de la guarida.	97
4.12. Diagrama de clases con la fachada, los DAOs y la representación en la base de datos de los monstruos y los huevos de monstruo.	99
4.13. Diagrama de clases con los VOs que encapsulan los detalles de los monstruos.	100
4.14. Diagrama de objetos donde hay un monstruo adulto con dos condicionantes, uno aumenta la felicidad y el otro la capacidad de enseñanza .	102
4.15. Diagrama de objetos con la representación de un atributo compuesto.	102
4.16. Diagrama de clases de las tareas.	104
4.17. Estructura de clases que organiza los aspectos relacionados con la configuración.	106
4.18. El patrón <i>Front Controller</i> en <i>Struts</i> .	109
4.19. <i>Layout</i> genérico de las pantallas durante el juego.	111
4.20. Méodos representativos de la clase <i>SessionManager</i> .	113
4.21. Gerarquía de acciones en el controlador de la aplicación.	114
4.22. Filtros de preprocesamiento definidos en <i>Thearsmonsters</i> .	116
4.23. Ejecución de una petición a través de los filtros de preprocesamiento.	117
4.24. Una ventana típica de la interfaz del juego, tiene pestañas que muestran diferentes contenidos dentro de la ventana.	119

4.25. Método para mostrar las descripciones internacionalizadas de los atributos en la vista.	126
4.26. Diagrama resumen de las interacciones necesarias para implementar el caso de uso de sugerir tareas mediante <i>Ajax</i>	134
5.1. Espacio <i>HSV</i> del color.	145
5.2. Comparación entre un párrafo sin espacios (izquierda) y otro que separa adecuadamente el texto de la imagen.	146
5.3. Ejemplo de buen diseño Web: Página principal de <i>Firefox</i>	147
5.4. Comparación de una imagen vectorial con una rasterizada.	148
5.5. Editando gráficos con el <i>GIMP</i> [1]. Prácticamente todos los elementos (<i>backgrounds</i> y <i>sprites</i>) de la interfaz web se han hecho con este programa.	149
5.6. Creación del gráfico vectorial de un monstruo con el <i>Inkscape</i> [2] a partir de su diseño en lápiz escaneado.	151
5.7. Editando la imagen de fondo del juego con <i>Photoshop</i> [3].	151
5.8. A la izquierda una captura del <i>Gears of Wars 2</i> . A la derecha del <i>Patapón</i> . Si en un juego no se puede crear un entorno gráfico realista, al menos debe contar con un buen diseño gráfico que sea estético, original y atractivo.	152
5.9. Algunos prototipos que ayudaron a definir el <i>layout</i> del juego. El estilo y la composición logradas se han ido refinando en cada uno, tratando de enfatizar y las ideas principales que debe expresar.	156
5.10. Captura de pantalla del juego durante una fase bastante avanzada del desarrollo. Ya puede apreciarse cómo va a quedar el <i>layout</i> final, y se puede comparar con los prototipos previos de la figura 5.9.	158
5.11. Vista de guarida del juego <i>Dig'n'fight</i>	161
5.12. Boceto de la vista de la guarida en el juego.	162
5.13. Vista final de la guarida. Se cambia el aspecto cavernoso por otro más urbano.	163
5.14. Algunos bocetos previos realizados para idear nuevas razas de monstruo.	165
5.15. Evolución del diseño y estilo gráfico de la raza de monstruo <i>Quad</i>	166
5.16. Evolución del diseño y estilo gráfico de la raza monstruo <i>Ubunto</i>	166

5.17. Proceso para crear los gráficos de los monstruos. A partir de un dibujo escaneado se crea el modelo vectorial, que después se convierte en mapa de bits para poder ser integrado en la página web.	167
5.18. Cría, adulto y anciano de la raza de monstruo <i>Electroserpe</i> . Se trata de monstruos peligrosos y efectivos en combate cuando desarrollan sus habilidades eléctricas.	168
5.19. Cría, adulto y anciano de la raza de monstruo <i>Quad</i> . Son los monstruos con más fuerza que hay en el juego.	168
5.20. Cría, adulto y anciano de la raza de monstruo <i>Ubunto</i> . Es la raza con más inteligencia.	168
A.1. Ejemplo de implementación del patrón arquitectónico MVC en <i>Java</i> . . .	178
A.2. Comparación entre los tres grandes sectores del desarrollo Web libre, expuesta por <i>Tim Bray</i> en la <i>PHP International Conference</i> en el año 2006.	180
B.1. Petición HTTP en el modelo de aplicaciones Web clásico.	184
B.2. Petición HTTP con el modelo de aplicaciones Web <i>Ajax</i> . Requiere el uso de <i>JavaScript</i> en el navegador.	185

Capítulo 1

Introducción

Índice general

1.1.	Visión general	1
1.2.	Objetivos del proyecto	2
1.3.	Fases del proyecto	3
1.4.	Descripción del contenido de la memoria	5

El objetivo principal de este proyecto es mostrar el proceso completo necesario para crear el juego *Thearsmonsters*. Se trata de un juego multijugador masivo basado en navegador, por lo que muchos aspectos de diseño e implementación están íntimamente ligados al desarrollo Web, sin embargo otros aspectos relacionados con la temática o la jugabilidad son comunes al diseño de videojuegos en general. Para el modelado del sistema se aplican técnicas y métodos habituales del diseño orientado a objetos.

1.1. Visión general

Thearsmonsters (1.1) es un juego en el que cada jugador debe gestionar una guarida en una ciudad subterránea donde los habitantes son monstruos pin-



Figura 1.1: Logotipo del juego *Thearsmonsters*.

torescos que realizan labores programables. Los monstruos tienen un ciclo de vida completo ya que deben aprender y mejorar habilidades, crecer, reproducirse y después morir, dejando paso a las nuevas generaciones. Por otro lado las guardias se mejoran constantemente y no se pueden destruir. Los jugadores deberán conseguir recursos, mejorar las características de la guarida y de sus criaturas para poder completar misiones más difíciles, prestar atención a la educación de las crías, atender a las relaciones sociales con otros jugadores y asegurar la estabilidad económica bajo sus dominios.

Si bien el juego puede clasificarse en la categoría de *MMORPG basado en navegador* (ver sección 2.3), el concepto particular del título *Thearsmonsters* es original del proyectando y forma parte del proyecto.

1.2. Objetivos del proyecto

Se trata de crear un juego y hacerlo realidad, no de que tenga unos gráficos impresionantes, de que haya muchas cosas o de que ofrezca infinitas posibilidades de movimiento. Simplemente debe ofrecer entretenimiento. Para lograrlo basta con un juego sencillo pero ambicioso, siempre teniendo en cuenta las limitaciones de un proyecto fin de carrera. Para desarrollar un juego competente en el mercado actual es necesario contar con un equipo de alrededor de cien empleados trabajando durante más de dos años. Por suerte es posible plantear otras posibilidades, como por ejemplo un juego pequeño de muestra, que se pueda implementar en una plataforma convencional como la Web y que utilice tecnologías validadas y bien probadas como *Java EE*.

Se trabajará con una idea original y se aplicarán las técnicas más convenientes para implementar las funcionalidades mínimas, dejando el diseño e im-

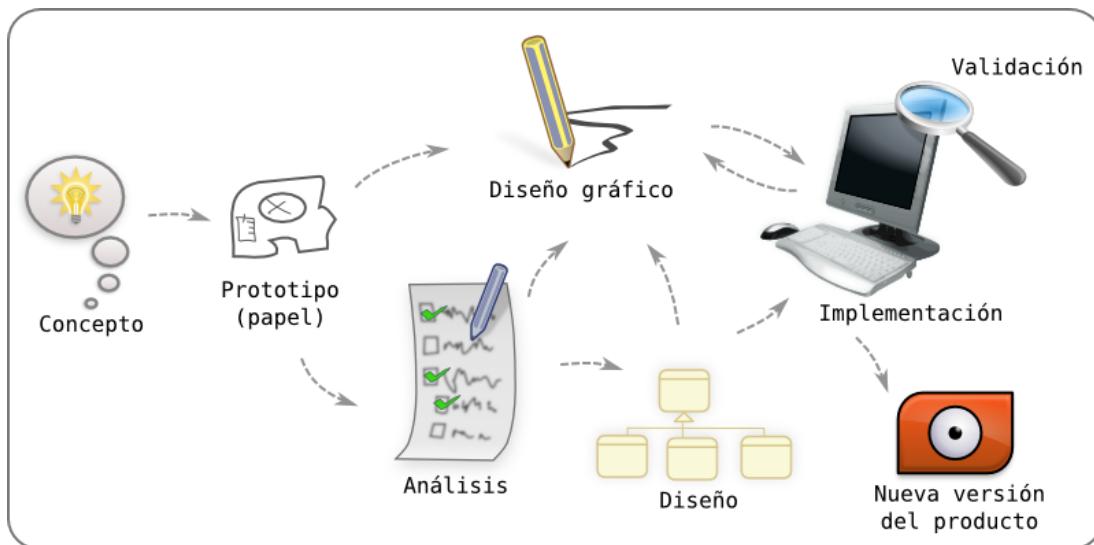


Figura 1.2: Esquema de las fases por las que pasa el proyecto

plementación preparados para futuras ampliaciones.

1.3. Fases del proyecto

Para realizar el proyecto con éxito será necesario seguir los siguientes pasos que se pueden ver esquematizados en la figura 1.2:

Diseño y concepto completo del juego: Antes de comenzar a trabajar hay que sentar bien las ideas del juego. Esta parte es más complicada cuanto mayor sea la originalidad, sobretodo en un juego multijugador, ya que es difícil predecir el comportamiento que tomará una comunidad de cientos o miles de usuarios.

Prototipado: es indispensable en un proyecto para el cual no se sabe muy bien lo que se quiere hacer. En el caso de un juego es muy importante tener alguna referencia que se pueda enseñar a posibles jugadores para que ayuden al diseñador a corregir errores potenciales a tiempo. Así mismo obliga a tener

un punto de referencia a la hora de fijar las bases de juego. En el caso de este proyecto se hicieron varias simulaciones de interfaz gráfica sobre papel, esquemas y representaciones de a vida de los monstruos.

Diseño de la interfaz gráfica: a medida que se va desarrollando el prototipo se va decidiendo como será el aspecto final del juego, sin entrar en detalles. Es bueno hacerlo ya en esta fase para que luego la parte gráfica se pueda parallelizar con la programación y además ayuda a dirigir el transcurso del proyecto. Más adelante, en las fases de análisis y diseño, hay ciertas dependencias que influyen en el aspecto gráfico final, por lo tanto esta fase se prolonga durante casi todo el ciclo de vida.

Análisis completo del juego: en esta fase se deciden las cosas que se van a incluir y las que no, se analiza el comportamiento de las ideas que han surgido, se conceptualiza el dominio utilizando UML y se evalúa el resultado obtenido.

Diseño completo del sistema: se comienzan a tener en cuenta aspectos de como se construirá un sistema que abra el dominio conceptualizado y se acercan la ideas previamente analizadas al lenguaje de la máquina, aunque no se hace estrictamente para una tecnología en particular. Para ello se utilizará también el lenguaje de modelado UML. Se definirá una arquitectura y un diseño general de todo el juego y después se seleccionarán las funcionalidades mínimas para implementar en la versión beta. El resto del modelo se utilizará más adelante, en futuras evoluciones.

Implementación de la versión beta del juego: se traduce una versión reducida del diseño anterior a un lenguaje de programación para poder ser ejecutado en una computadora.

Validación de la versión beta y reajustes en el modelo: realización de pruebas para asegurar que todo va bien encaminado. Si se encuentra algún problema se hacen los reajustes en el modelo, de forma iterativa.

Introducir funcionalidades para crear la versión 1.0: que no es todo el diseño anteriormente creado, sino una versión mínima del juego en la que se pueda jugar. A ojos del jugador será bastante incompleta, aunque para realizarla será necesario cubrir casi todas las funcionalidades del diseño. A partir de este punto y siguiendo las especificaciones del análisis y del diseño completos debería ser sencillo añadir nuevos contenidos (más monstruos, tipos de salas, tipos de batallas ...) para dotar al juego de mayor riqueza desde el punto de vista de la experiencia de usuario.

1.4. Descripción del contenido de la memoria

Se comienza con la *Contextualización*, que introduce las disciplinas más importantes (desarrollo Web, diseño de videojuegos) y el segmento de mercado donde se va a introducir el juego *Thearsmonsters*, teniendo en cuenta otros juegos de éxito en la actualidad.

En el capítulo *Concepto y análisis* se describe formalmente la idea original, se hace el análisis de requisitos, y se establece el modelo conceptual.

En el capítulo *Diseño e Implementación* se especifican los conceptos extraídos del análisis conforme a las tecnologías seleccionadas para realizar la implementación. A medida que se muestra el modelado y el planteamiento de cada concepto, se ilustra con algún ejemplo que se acerca más a la implementación realizada (diagrama de secuencia detallado, trozo de código, etc.).

Debido a la gran importancia que toma el diseño gráfico en el desarrollo de este proyecto se hace el capítulo de *Interfaz Gráfica*, en el que se explican las bases del diseño artístico junto con los procesos y las herramientas utilizadas para llevarlo a cabo.

Por último, la *conclusión* es un conjunto de opiniones personales del proyectando, desde un punto de vista global.

Capítulo 2

Contextualización

Índice general

2.1. Desarrollo web	8
2.2. Diseño de videojuegos	9
2.3. Juegos MMOG	11
2.4. Juegos <i>MMORPG</i> basados en navegador	15
2.5. Análisis de mercado	17
2.6. Juegos de referencia	19
2.6.1. Ogame	20
2.6.2. Travian	25
2.6.3. World Of Warcraft	28

El proyecto *Thearsmonsters* se puede clasificar con las etiquetas “desarrollo de una aplicación Web” y “diseño de videojuegos”. Lo primero es debido al despliegue tecnológico, que está basado enteramente en la Web, y lo segundo es porque un porcentaje muy alto del trabajo realizado consistió en definir y analizar las funcionalidades del juego, teniendo en cuenta a los usuarios potenciales del mismo. No se puede caracterizar como “programación de videojuegos” ya que

hoy en día este ámbito suele referirse al manejo de herramientas de modelado tridimensional y programación de aplicaciones gráficas cliente en tiempo real.

Thearsmonsters es un juego del tipo *MMORPG*¹ basado en navegador de estrategia y gestión económica, aunque también tiene un importante apartado de lucha.

En este capítulo se procede a explicar con más detalle cada uno de estos conceptos. También se hace un pequeño análisis de mercado y se analizan algunos juegos de éxito en la actualidad.

2.1. Desarrollo web

El desarrollo web es un título algo arbitrario para el conjunto de tecnologías de software del lado del servidor y del cliente que involucran una combinación de procesos de base de datos con el uso de un navegador en Internet a fin de realizar determinadas tareas o mostrar información.

Tradicionalmente un software departamental o incluso un ambicioso proyecto corporativo de gran envergadura es desarrollado en forma *standalone*, es decir, usando lenguajes ya sea compilados (*C*, *C++*, *Delphi*), semicompileados (*.Net*, *Mono*, *Java*), o interpretados (*Ruby*, *Phyton*) para crear tanto la funcionalidad como toda la interfaz de los usuarios, pero cabe perfectamente un desarrollo orientado a Web para dichos propósitos, siendo más homogéneo y multiplataforma, y dependiendo de las tecnologías utilizadas, más rápido y robusto tanto para diseñar, implementar y validar, como para su despliegue y explotación una vez terminado. Esto también se cumple en el mundo de los juegos multijugador como veremos más adelante.

Aunque en el caso de este proyecto todo ha sido realizado prácticamente por la misma persona, funcionalmente existen varios roles en el desarrollo Web. El *desarrollador Web*, que es quien realiza esta labor, normalmente sólo se preocupa

¹MMORPG son las siglas del inglés *Massively Multiplayer Online Role Player Game*, que significan Juego Online Multijugador Masivo de Rol. Este término se trata en la sección 2.3

por el funcionamiento del software, siendo tarea del *diseñador Web* el preocuparse del aspecto final (*layout*) de la página y del *webmaster* el integrar ambas partes. En ocasiones el *webmaster* también se encarga de actualizar los contenidos de la página.

Más referencias sobre desarrollo y diseño Web en [26], [20], [33] y [29].

2.2. Diseño de videojuegos

El desarrollo de videojuegos es la actividad por la cual se diseña y produce un videojuego, desde el concepto inicial hasta el juego en su versión final, el producto terminado.

Ésta es una actividad multidisciplinaria, que involucra profesionales de la informática, el diseño, el sonido, la actuación, etc.

El desarrollo de un videojuego generalmente sigue el siguiente proceso:

1. Concepción de la idea del juego
2. Diseño
3. Planificación
4. Producción
5. Validación
6. Mantenimiento

El proceso es similar a la creación de software en general, aunque difiere en la gran cantidad de aportes creativos necesarios: música, historia, diseño de personajes, niveles, etc. El desarrollo también varía en función de la plataforma objetivo (PC, móviles, consolas ...), el género (estrategia en tiempo real, rpg, aventura gráfica, plataformas ...) y la forma de visualización (2D o 3D).

El diseño es la fase en la que se detallan todos los elementos que compondrán el juego, dando una idea clara a todos los miembros del grupo desarrollador acerca de como son. Entre estos elementos:

Diseño de Arte: Abarca la historia, el sonido, la interfaz y los gráficos del juego.

Historia: Forma en que se desenvolverán los personajes del juego y la manera en la que la historia del mundo representado avanza. No todos los juegos tienen historia.

Sonido: Detallada descripción de todos los elementos sonoros que el juego necesita para su realización. Voces, sonidos ambientales, efectos sonoros y música.

Interfaz: Es la forma en que se verán los elementos *GUI* (*Graphical User Interface*, que significa *Interfaz Gráfica de Usuario*), mediante los cuales el usuario interactuará con el juego.

Gráficos: Dependiendo de si el juego es 2D, 2.5D o 3D, en este apartado se deben especificar los *sprites*, *tiles* y modelos 3D a utilizar. Generalmente esta fase incluye un desarrollo conceptual y una especificación de las características de los modelos.

Diseño de mecánicas: Es la especificación del funcionamiento general del juego.

Es dependiente del género y señala la forma en que los diferentes entes virtuales interactuarán dentro del juego, es decir, las reglas que rigen éste.

Diseño de programación: Describe la manera en que el videojuego será implementado en una máquina real (PC, consola, móvil, etc) mediante un cierto lenguaje de programación y siguiendo una determinada metodología. Generalmente en esta fase se generan diagramas UML que describen el funcionamiento estático y dinámico, la interacción con los usuarios y los diferentes estados que atravesará el videojuego como software. Esta fase será ampliamente comentada en el capítulo dedicado al diseño de esta memoria.

Referencias sobre el diseño de videojuegos en [24], [25] y [10].



Figura 2.1: Ejemplo de MMORPG tradicional: los jugadores se agrupan para charlar en una plaza del juego *Guildwars*.

2.3. Juegos MMOG

El término MMOG viene de las siglas inglesas *Massively Multiplayer Online Game*) y significa juego en línea multijugador masivo, refiriéndose a un tipo de juegos en concreto. Este género difiere de un juego en línea multijugador no masivo en que éstos últimos tienen un número limitado de jugadores, es decir, los MMOGs están preparados y elaborados de tal manera que admiten cualquier número de jugadores simultáneos, aunque en la práctica viene limitado por la conexión y capacidad del servidor.

Empezaremos distinguiendo dos grandes grupos de juegos multijugador en línea por la tecnología que utilizan.

El primer grupo es el de los MMOGs que tienen una parte cliente que se conecta a uno de los servidores disponibles, es decir, lo que se denomina técnicamente modelo cliente-servidor. Los clientes son programas específicos instalados

en el ordenador del usuario que se encargan de representar la porción del mundo visible por el personaje y de enviar las acciones del personaje de vuelta al servidor. Es en el servidor donde suceden realmente los acontecimientos del mundo virtual, y donde además residen los datos críticos (por ejemplo el dinero que tiene un determinado personaje) para que estos no puedan ser fácilmente falseados.

Los clientes, en este modelo, son programas fundamentalmente tontos (desde el punto de vista de la lógica del dominio que implementan). Lo cual no significa que sean pequeños o simples, de hecho, la tendencia actual es que sean juegos en 3D con gran nivel de detalle gráfico, lo que suele implicar una gran exigencia de recursos al ordenador cliente, equivalentes a los de cualquier otro videojuego.

El gran problema de esta arquitectura cliente-servidor son las actualizaciones. Los programas suelen tener siempre fallos, pero en el caso de los MMOGs además de corregir éstos, también hay que evitar todos los *bugs* o trampas que se vayan descubriendo porque esto puede desequilibrar el juego desfavoreciendo a muchos jugadores que pueden dejar de pagar sus cuotas mensuales. Los *MMOGs* tienen una largo ciclo vital, frecuentemente de años, por lo que deben evolucionar en el tiempo. Siempre hay que ampliar los escenarios, las características, los objetos, etc. Esto se traduce en sucesivas expansiones que exigen más y mayores cambios en el cliente, además de en los servidores.

El segundo grupo de juegos multijugador en línea son los denominados *basados en navegador* o *browser games*, que están muy de moda y alguno de ellos son masivos, esto es *MMOGs*. Esto hace referencia a que los juegos tienen algún tipo de interfaz Web o están basados en alguna tecnología basada en un navegador web. Naturalmente el hecho de basarse en el navegador limita fuertemente la interactividad gráfica del juego (aunque hay algunos que lo suplen con tecnologías como *Flash*). Hay quien critica que no son estrictamente videojuegos sino juegos de llenar casillas con números. Es cierto que la mayoría de *MMOGs* de este tipo son juegos de que funcionarían en modo texto, aunque cada vez hay más excepciones.

La ventaja de esta clase de *MMOGs* no es sólo el hecho de tener un cliente



Figura 2.2: Ejemplo de MMORPG para navegador: alineación de un equipo en *Hattrick*. El juego se visualiza y controla desde un navegador Web como por ejemplo Firefox.

universal (el navegador Web) para acceder a ellos, sino que de un golpe se termina con el problema de las actualizaciones y las expansiones. La Web siempre está actualizada, evitando de este modo todos los problemas y dolores de cabeza que suponen los sistemas cliente-servidor, en los que el cliente es una aplicación *ad-hoc*.

En cualquier caso, ya hay una primera división entre *MMOGs* cliente-servidor (los más habituales), y los basados en Web (en fuerte ascenso). Otra división fundamental es la de los géneros², aunque como los primeros juegos del género reconocidos como multijugadores masivos son los RPGs en línea, lo normal es que *MMORPG* sea el término utilizado para cualquier género³ de lo que, siendo más precisos, se debería englobar bajo el paraguas *MMOG*.

Por ejemplo, el juego *EVE-ONLINE* (de naves en el espacio) se suele encuadrar dentro de los *MMORPGs* (los jugadores asumen roles como los de minero, pirata, policía o comerciante) pero el pilotaje de la nave lo convierte en un simulador aéreo, y las batallas, en un auténtico arcade en 3D, o incluso en un juego táctico (cuando un grupo de naves se enfrenta a otra). Si a eso le añadimos las políticas de alianzas entre las diferentes confederaciones, las zonas de influencia, la diplomacia, las guerras, traiciones, robos, etc, de repente pasamos a otro nivel que es el estratégico/político. Y eso sin hablar del sistema tecnológico/económico, que es también otro tipo de estrategia. O sea, que en un mismo juego se combinan en realidad elementos de géneros muy diferentes, y con gustos muy diversos, lo cual puede crear una mezcla explosiva y difícil de predecir. De hecho, incluso en un *MMORPG* que fuera puramente un juego de combate entre naves, sin componente rol o economía, seguramente se produciría un fenómeno estratégico en el mismo, a través de pactos y alianzas. Así que cuando hablamos de géneros dentro de un *MMOG*, tendremos que tener siempre en cuenta

²Si en el videojuego tradicional se pueden encontrar juegos de acción, de estrategia o de rol, en los *MMOGs* están los *MMORPGs* (es decir, los *MMOGs* de rol), los *MMOFPS* (*First Person Shooters*) o los *MMOG* estratégicos (en tiempo real o basados en turnos), entre otros géneros

³En cierta manera, los *RPGs* en línea han recogido elementos más allá del propio concepto de *RPG*, para convertirse en lo que se han denominado mundos virtuales

la mezcla de características que abunda en la mayoría de ellos. Debido a esta complejidad y también a la creciente popularidad del género, cada vez hay más psicólogos, sociólogos y antropólogos que estudian las acciones e interacciones de los jugadores de tales mundos virtuales.

Muchos pequeños equipos de programadores y artistas individuales han intentado crear su propio *MMORPG*. Un proyecto de esta clase consta de entre tres o cuatro años de media, y los desarrollos *amateurs* pueden incluso superarlo debido a la falta de tiempo, mano de obra o dinero. Además, el coste para mantener los servidores puede ser la razón más importante para terminar abandonando el proyecto. Los *MMORPGs* gratuitos generalmente se basan en el tiempo libre invertido por un pequeño grupo de programadores, artistas y diseñadores del juego.

2.4. Juegos *MMORPG* basados en navegador

Los *MMORPGs* basados en navegador o *browser games* son un caso especial de los *MMORPGs*, generalmente más sencillos que sus contrapartidas basadas en cliente, normalmente basándose en un juego con estrategias más simples, del tipo “construye un gran ejército, y entonces ataca a otros jugadores para robarles recursos”.

Es evidente pensar que un juego basado en navegador no puede tener todas las funcionalidades que otro basado en cliente. Las tecnologías que abordan la web están pensadas para hacer páginas clásicas, que normalmente no requieren de gráficos 3D ni de sofisticadas reglas físicas. Aunque esto no es del todo cierto, ya que actualmente ya existen tecnologías para poder conseguir estos mismos resultados (véase *Java* o *Flash*), aunque son muchísimo menos eficientes y necesitan un tiempo de desarrollo mucho mayor.

Teniendo en cuenta esto, se puede acotar el dominio de los juegos para navegador a juegos representacionales basados en formularios que hay que llenar y en pantallas que muestran información, además de algunas animaciones, gráficos

y sonido muy puntuales. Dentro de este tipo de juegos el formato Web presenta muchas ventajas. Para empezar se libra al usuario de realizar instalaciones, ya que lo único que tienen que hacer es conectarse a la URL del juego y todo el código necesario será interpretado por el navegador (con los plugins necesarios para los casos de Flash o Java, que se suelen instalar automáticamente si no se dispone de ellos). Hacer que el juego sea multijugador es muy sencillo porque casi todas las tecnologías web disponibles en la actualidad permiten conectarse fácil y eficientemente a una base de datos centralizada, donde se almacena el estado del mundo virtual. Otra ventaja es que todos estos procedimientos están muy bien estudiados y probados, permitiendo minimizar el riesgo de fracaso en el proyecto.

Por parte de los usuarios, un juego basado en navegador no tiene por qué tener gráficos espectaculares, ya que no es algo que estén esperando de antemano. Sin embargo, intentar hacer un juego basado en cliente y no proporcionar un nivel gráfico adecuado puede provocar incertidumbre e insatisfacción. Una interfaz Web es suficiente cuando solamente queremos representar estados y dotar al jugador de los medios necesarios para modificarlos. Lo difícil en este caso es hacer que los jugadores sepan en todo momento donde están y qué están haciendo, mantener la sensación de estar interactuando en un mundo virtual con otros jugadores es complicado cuando no se pueden mostrar animaciones y escenarios. La información debe estar bien actualizada y representar correctamente el estado de los demás jugadores.

Este tipo de juegos permite un tiempo de desarrollo mucho más reducido, por ello últimamente han aparecido multitud de títulos en este género, la mayoría de mala calidad y muy repetitivos, aunque también se pueden encontrar interesantes variaciones de los temas habituales, donde se tratará de situar la temática de nuestro juego *Thearsmonsters*.

2.5. Análisis de mercado

Lejos de intentar realizar un análisis de mercado exhaustivo, se explicarán brevemente los tipos de jugadores que se pueden encontrar y se describirán los aspectos relevantes del segmento de mercado que los abarca.

Por desgracia, el mercado de los MMORPGs basados en navegador está actualmente en oferta, debido que hay demasiados juegos de este tipo⁴. De todas formas hay cientos de miles de jugadores potenciales en Internet, que están aburridos de jugar siempre al mismo tipo de juegos, o que simplemente nunca han probado un juego de este tipo. La estrategia para llamar su atención en este caso puede ser, o bien mejorar los gráficos, tecnología y jugabilidad de los títulos existentes, o bien crear algo diferente a lo habitual con respecto a temática y ambientación, que es lo que pretende este juego.

Hay jugadores que prefieren hacer misiones individuales, aunque estén en un juego multijugador, otros jugadores prefieren relacionarse y disfrutan personalizando y mostrando sus personajes, otros prefieren comerciar y hacerse virtualmente ricos y también los hay que prefieren ante todo luchar y competir para demostrar que son los mejores.

Hay muchos matices en los tipos de jugadores, aunque principalmente se diferencian tres:

Los *Casual players* o jugadores casuales, que son personas que nunca han probado este tipo de juegos y están aprendiendo. Lo normal es que prefieran jugar sin presiones, de forma cooperativa y amistosa.

Los *Hardcore players*, que juegan compulsivamente y siempre están conectados. La mayoría de juegos que hay actualmente en Internet están enfocados básicamente a este tipo de jugadores, ya que son los que producen más beneficios. Para hacer un juego así está bien permitir que haya muchos

⁴Uno de los principales motivos es que aunque el juego sea gratuito para los usuarios, se puede ganar dinero solamente por incluir *banners* publicitarios en el sitio Web que los alberga

combates entre ellos, que se puedan robar unos a otros y que los fuertes tengan poder sobre los débiles.

Los *Noobs* o novatos, que son todos en general, cuando están aprendiendo. Este tipo de jugadores necesitan que la curva de aprendizaje del juego sea suave, tener algún tipo de protección contra los jugadores experimentados y disponer de accesibilidad a una comunidad que les apoye y les ayude en la difícil tarea de aprender no solo las normas del juego, sino todos los convenios sociales que surgen alrededor del mundo virtual. Muchos de estos jugadores nunca llegan a más, y acaban por convertirse en "pasto" para los demás. Ello se debe a que normalmente hacerse una cuenta de prueba es gratis, y como no consiguen comprender o no disfrutan del juego, se van sin más, dejando su personaje o imperio sin atención, facilitando el saqueo de los demás jugadores.

El juego desarrollado en este proyecto, *Thearsmonsters* está enfocado sobre todo a los jugadores pacíficos, impidiendo que se puedan hacer ataques agresivos entre jugadores, facilitando todo lo posible el aprendizaje del entorno y enfocando el avance del juego desde un punto de vista individualista. Aunque la mayoría de jugadores que conocen este tipo de juegos son *hardcore players*, ya hay demasiados títulos que están íntegramente diseñados para satisfacerlos. De este modo se pretende también llamar la atención del género femenino, que está poco explotado, y así introducir el juego en un mercado más pequeño, pero con más demanda.

A continuación, un pequeño análisis *DAFO*⁵ simplificado, que permite ver con claridad cuales son las ventajas y los riesgos de llevar a cabo el despliegue de este producto.

- Debilidades
 - Mucha competencia (mercado muy atomizado).

⁵El análisis DAFO es una metodología de estudio de la situación competitiva de una empresa en su mercado y de las características internas de la misma, a efectos de determinar sus Debilidades, Amenazas, Fortalezas y Oportunidades

- Baja fidelidad de los usuarios hacia la mayoría de este tipo de juegos.
 - Existencia de competidores muy maduros y con alta cuota de mercado.
- Amenazas
- Aparición de competidores continuamente.
 - Mala imagen de las campañas publicitarias que se suelen hacer, los usuarios pueden considerarlas *spam*.
- Fortalezas
- Temática muy original, y sin embargo, dentro de las tendencias actuales.
 - Nueva visión y modalidades de juego.
 - Situación en un segmento (nicho “casual”) distinto al de los competidores directos.
 - Al contrario que la mayoría de competidores, este juego esta indicado para casi cualquier tipo de público.
- Oportunidades
- Afloran los usuarios que buscan juegos gratuitos (ocio barato debido a la crisis).
 - La poca fidelidad puede servir para atraer jugadores a nuestro nicho.

2.6. Juegos de referencia

Existe una gran multitud de juegos similares a *Thearsmonsters* en la actualidad. En el apéndice C se muestra una lista de MMORPGs basados en navegador que sirven como referencia para ver cómo se encuentra el sector en este momento.

De esta lista se pueden extraer fácilmente conclusiones acerca del estilo dominante en este tipo de juegos, las pocas categorías que hay y la asombrosa falta de originalidad.

Muchos juegos han influido en el diseño de *Thearsmonsters*, pero hay unos pocos en concreto a los que se ha prestado más atención, que marcan tendencia, y que de alguna manera consiguen ser masivamente visitados por miles o incluso cientos de miles de jugadores diariamente. Por ello merece la pena analizar las características que los hacen tan especiales.

2.6.1. Ogame

Básicamente, es la definición de juego para navegador. Ha marcado un antes y un después por lo menos en el entorno hispanohablante, ya que es el primer juego realmente masivo que se ha podido disfrutar en español. Este juego de “estrategia espacial” hizo furor a mediados del 2005, y ha tenido enganchados, incluso sin dormir, a muchos internautas. La fiebre ya ha pasado, pero las consecuencias durarán. [15]

La mayoría de juegos basados en navegador actuales imitan la forma de juego del *Ogame*, aunque realmente *Ogame* no sea juego muy innovador, ya que es una copia bastante descarada de otro juego anterior llamado *Planetarion*, el cual originalmente era gratuito y fue masivamente abandonado cuando se hizo de pago. A su vez, *Planetarion* y otros juegos similares están inspirados en un juego más antiguo al que se jugaba por correo electrónico llamado *VGA Planets*. Inspirándose unos en otros, al final existen tantos juegos de estrategia espacial basados en navegador que casi se podría hablar de un subgénero aparte. Por ello hay que tener en cuenta que en el análisis que se va a hacer a continuación, aunque se analice el *Ogame*, en realidad se está describiendo la mayoría de juegos basados en navegador que hay en la actualidad.

En *Ogame*, cada jugador es el emperador de un planeta, y debe regir los destinos de su imperio expandiéndolo hasta dominar el universo. En realidad, un universo es el equivalente al concepto de partida, y también de servidor: todos los



Figura 2.3: *Ogame*, juego espacial basado en navegador

jugadores que se crean una cuenta en el mismo universo, participan en el mismo tablero de juego. Solo se permite una cuenta por persona y universo. Si se identifica a un jugador con más de una cuenta, se considera trampa y los administradores lo eliminan de la partida. Cada cierto tiempo se van abriendo nuevos universos tanto para acomodar a más jugadores, como para que los jugadores empiecen más o menos al mismo tiempo, y tengan por lo tanto las mismas oportunidades. Cuando se prueba el juego por primera vez, parece sorprendente que algo tan simple pueda causar tanto revuelo, ya que pronto se descubre que el sistema de juego es muy sencillo, simplemente se trata de mejorar el planeta inicial para crear más naves para conquistar más planetas y así sucesivamente.

Una característica fundamental del juego es que todos los edificios y tecnologías funcionan por niveles. Por ejemplo, la producción de metal se consigue con las minas de metal, y la capacidad de extracción de las mismas depende de su nivel. Con las tecnologías pasa lo mismo, cada nivel cuesta el doble que el anterior, sin límite definido. El juego no termina nunca, al contrario de otros MMOGs similares como *Planetarion*, donde la partida se reiniciaba cada 2 meses. Esta característica se aplicará también al sistema de juego de *Thearsmonsters*, ya que así no se limita la duración de la experiencia de juego.

El crecimiento del coste de subida de niveles es exponencial. En los niveles más bajos, para recuperar los recursos gastados en subir de nivel una mina, son necesarios unos minutos, pero por encima del nivel 20, por ejemplo, son necesarias semanas, incluso meses de extracción. Con las tecnologías pasa lo mismo: a partir de ciertos niveles resulta simplemente inalcanzable el coste de aumentar otro nivel más. Lo mejor es subir los niveles más bajos lo antes posible, hasta el nivel óptimo, y producir a ese nivel el máximo tiempo posible.

Con esa premisa, es obvio que el juego se convierta en una carrera contrarreloj

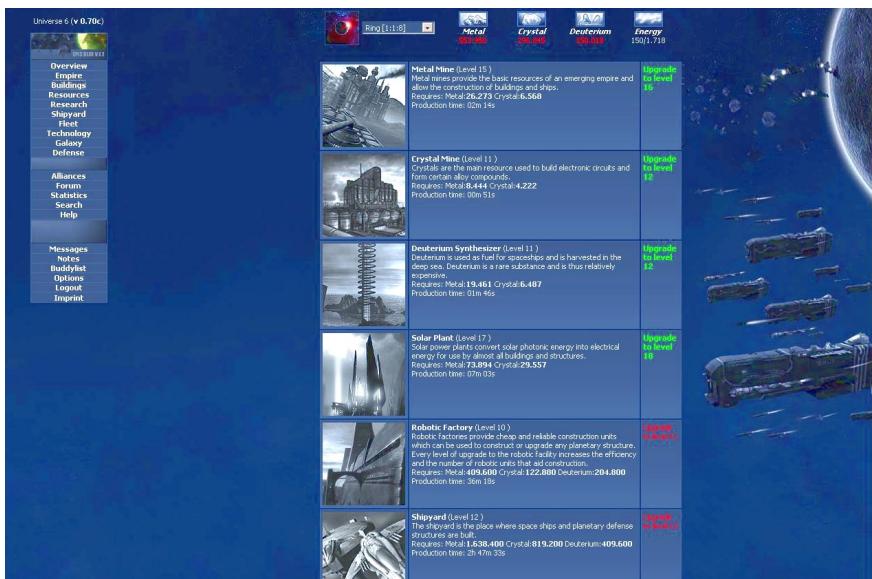


Figura 2.4: Captura de pantalla de *Ogame*: Edificios de un planeta.

por conseguir recursos. Cuanto antes tengas las unidades necesarias para subir un nivel de minas o tecnologías, antes podrás producir más que el resto, tener más naves, más tecnología, etc. Sin embargo el método más rápido para crecer no es esperar a producir tú mismo esas cantidades, sino robárselas al resto de los jugadores, y en este punto es donde está la clave del juego, el pillaje.

Al principio, con la poca tecnología de la que se dispone sólo se podrán hacer primitivas naves de transporte y pequeños cazas. Pero con esas naves ya se puede atacar cualquier planeta que no tenga defensa, y robarle casi todos los recursos que tenga. Por eso la clave es comenzar cuanto antes en un universo nuevo, para tener las primeras naves y las minas más antiguas. Después los jugadores que comiencen desde el principio podrán ser controlados y dominados porque siempre serán inferiores.

Aun así, el robo de recursos es sólo el aperitivo del pillaje. A partir de cierto nivel viene la segunda parte, porque las flotas destruidas dejan escombros que pueden ser reciclados para producir más recursos. Esta es la sorpresa de los jugadores noveles: una mañana se conectan y descubren que alguien con muchas

o muchísimas más naves que ellos han atacado sus planetas y destruido todas sus flotas, recogiendo todos los escombros y sumándolos a sus reservas. Destruir flotas se convierte no en un objetivo político, sino en el camino fácil y más rápido para crecer, y los ataques sólo son importantes por sus ganancias netas.

Hay que fijarse en la ironía del asunto: se construye una flota de naves para defender los planetas que se han conquistado y poder atacar a otros, pero resulta que esa flota es justo lo que codician los demás. El elemento de fuerza es a la vez la mayor debilidad, y el juego se convierte en un continuo “cazar mientras no te seas cazado”. No es cuestión de ética o política, no se destruye por llevarse mal con alguien, como parte de un rol o por cualquier otra causa. El motivo es la pura y simple depredación, y la única defensa es procurar no ser una pieza de provecho.

Además, debido al componente de tiempo real (el juego siempre está funcionando en el servidor, aunque no se esté conectado), no se puede bajar nunca la guardia. De ahí proviene la famosa adicción: de la continua atención que hay que prestarle, siempre pendiente de atacar y esquivar ataques. Y mientras, no puede haber descansos ni distracciones, puesto que muchos jugadores por detrás podrían sobrepasar todo lo que se haya conseguido.

Con respecto a los clanes o alianzas de jugadores, existen dos tipos fundamentales: Los *Uber Guilds*, que son clanes formados por jugadores muy poderosos y los *Zerg Guilds*, que son, en cierta manera, el extremo contrario, clanes cuyo poder reside en su número. Los *Zerg Guilds* practican ataques masivos en los que resultan ganadores simplemente por su superioridad numérica, aunque sus miembros sean novatos de muy bajo nivel.

Ya desde los primeros MMORPGs, los diseñadores se dieron cuenta que el *zerging* era una táctica muy poderosa, tanto para acabar con poderosas bestias, destinadas para jugadores de mayor nivel, como para acabar con estos mismos jugadores poderosos. Así que decidieron utilizar técnicas que evitasen el *zerging*. Por ejemplo, las zonas de monstruos suelen residir en estrechas mazmorras, que no permiten el paso de muchos jugadores a la vez. Además los monstruos poderosos

suelen rodearse de muchos monstruos menores. Y también se fomentan los llamados ataques de área, que reparten el daño en una zona, afectando a múltiples jugadores a la vez.

Ogame tiene sus propios mecanismos para evitar este tipo de ataques. Por ejemplo, el número de jugadores que se unen a una batalla es limitado y además no se pueden beneficiar económicamente por pertenecer a grupos grandes. En realidad, lo que suele pasar en los universos de *Ogame* es que hay un *Uber Guild* de unos pocos jugadores muy poderosos que comandan la clasificación y que generalmente arrasan con todos los demás.

Otra de las características más decepcionantes de *Ogame* es su vertiente económica/comercial. En principio, aparte de las naves militares, hay naves de transporte con las que se podría intercambiar unos recursos por otros con los demás jugadores. El intercambio está limitado y vigilado por los *webmasters*, pero el problema no es ese. El problema fundamental es que el comercio no aporta ninguna ventaja a quienes lo practican porque se limita al intercambio de los 3 recursos fundamentales (metal, cristal y deuterio). No se pueden por ejemplo, producir naves para otros. Pero es que además, no hay forma de especializarse en producir uno de ellos en concreto. Cualquiera puede producir de los tres sin problemas, y de hecho, por su carácter exponencial, hay un nivel óptimo a partir del cual no merece mucho la pena subir. Y en realidad estos niveles están bastante equilibrados, así normalmente todo el mundo es bastante autosuficiente de todos los recursos. Y si no lo es, lo terminará siendo.

El único comercio que triunfa es el de la información. Saber donde hay flotas es siempre el interés de los depredadores, y manejada con astucia esa información puede ayudar a quitar algunos competidores. Por desgracia, la oferta de información es abundante, por lo que se devalúa bastante, y reduce su efectividad.

Sin nada en exclusiva, no hay comercio. Y sin comercio, no se pueden crear intereses, con lo que no hay nada que ofrecer que pueda motivar pactos y alianzas. El mecanismo que funciona en la sociedad, por el cual es más beneficioso para ambas partes la cooperación que la confrontación (lo que se llama en teoría de

juegos “el dilema del prisionero”) no funciona en *Ogame*, convirtiéndolo en un simple juego táctico, de escasa estrategia, de batallas en el que la propia dinámica absorbente del mismo arrastrará al jugador en una espiral competitiva hasta que se termine aburriendo.

Con sus pros y sus contras, este sistema de juego es muy adictivo y es el que usan la mayor parte de juegos basados en navegador. En el sistema de juego de *Thearsmonsters* se intentan evitar los contras, por ejemplo, no se permite atacar a otras guardias ni robar recursos. También se intentan favorecer los aspectos más débiles en *Ogame*, como la especialización en un tipo de servicio que permita producir beneficios económicos. Más adelante, en el capítulo de análisis, se explicará profundamente el nuevo sistema de juego ideado.

2.6.2. **Travian**

Sigue la filosofía de los juegos tipo *Ogame*, pero ambientado en un mundo donde las aldeas de Romanos, Galos y Germanos luchan entre sí. Cada jugador debe elegir la raza al comienzo del juego, debiendo después ajustar su estrategia a las características particulares de cada una. También se diferencia del *Ogame* en que el árbol de tecnologías es finito, y sobretodo en que puede destruirse absolutamente todo lo que se construya (en *Ogame* solo se destruyen las naves, no los edificios de los planetas), aunque en general prima el aspecto defensivo ya que se dan más facilidades para defender una aldea que para atacarla. Pese a las diferencias, en realidad estamos hablando del mismo tipo de juego, así que en vez de “planetas y naves”, hay “aldeas y soldados”, en vez de recursos como “metal, cristal o deuterio”, hay “madera, barro, hierro y cereales”, y en vez de “Fábricas de Robots” hay “Edificio principal” para rebajar tiempos de construcción. Los juegos persistentes en tiempo real conducen a estar pendientes a todas horas de si se recibe algún ataque, y eso tampoco es una excepción en *Travian*. Esto es un ejemplo de la similitud que suele haber entre los diferentes juegos basados en navegador.

Sin embargo hay algunas peculiaridades que no se pueden pasar por alto, al



Figura 2.5: Travian, juego de batallas medievales para navegador

fin y al cabo *Travian* también es un juego que goza de gran éxito y que tiene sus propios seguidores. Por ejemplo, aunque se puedan destruir las aldeas e incluso llegar a exterminar a un jugador y hacerle empezar casi de cero, no es algo que se haga indiscriminadamente, es más, quien lo hace siempre busca la forma de justificarlo. Esto es porque no se obtiene ningún beneficio en las batallas (no hay escombros reciclables como en *Ogame*), por lo que entre el colectivo de jugadores ha aparecido la conciencia de que el ataque destructivo es malvado. De todas formas es necesario menguar las defensas de la aldea enemiga para poder robarle recursos. Otra buena idea es la inclusión de objetos únicos, que permiten al jugador que los posee tener extras como doble producción de alimentos o más ataque para todas las unidades. Estos objetos son un bien muypreciado y escaso, que motiva a los jugadores más expertos a luchar para conseguirlos o mantenerlos en su posesión. También es fácil deducir que el apoyo de otros jugadores es fundamental, tanto para atacar, desgastando ejércitos con un mayor número de ataques, como para defender, acumulando defensas aliadas en la misma aldea. Es por ello que la diplomacia importa mucho, y esto es una propiedad que conviene maximizar, ya que en este tipo de juegos tanto los aliados como los enemigos son personas con las que se pueden establecer relaciones muy diversas y absorbentes. En *Travian* también se pueden hacer alianzas, aunque estas están limitadas en cuanto al número de participantes por un edificio llamado embajada, que hay que mejorar mucho si se desea tener un gran número de participantes (esto es otra forma de evitar el *zerging*, es decir, el que haya alianzas con muchísimos jugadores aunque sean de poco nivel). Como es difícil aumentar el número de



Figura 2.6: Representación de las aldeas en *Travian*. El crecimiento de la aldea es tangible.

participantes en una sola alianza entonces aparecen los pactos entre alianzas y así se van creando áreas de influencia, que están dominadas por uno o por otro pacto de alianzas.

Travian presenta muchos aspectos interesantes, sin embargo el que mayor influencia puede tener sobre el desarrollo de este proyecto es el de la interfaz gráfica. Aunque *Travian* se trate de un juego para navegador, es muy intuitivo saber qué está sucediendo en cada momento. Los menús son amigables y visualmente atractivos. El hecho de que el juego funcione sobre la web no interfiere demasiado en la jugabilidad, utiliza técnicas *Ajax* en lugares puntuales, que son un formidable acuerdo entre funcionalidad y facilidad de desarrollo, completamente aptos para ser aplicados a este proyecto. El estilo gráfico es simple y cómico,

lo cual demuestra una vez más que lo complicado debe ser el diseño, no hacer dibujos espectaculares llenos de detalles que pueden no encajar bien en el sitio al que pertenecen. Además el concepto de “Aldea de Galos” es similar a nuestra “Guarida de Monstruos”, y la representación de la aldea en *Travian* es más que satisfactoria, como puede observarse en la figura 2.6. Incluso existen los llamados *packs gráficos*, que permiten cambiar el aspecto de la aldea, por ejemplo, a un lugar nevado con luces de navidad. Todo ello ayuda al jugador a situarse en el ambiente deseado, a personalizar su presencia en dicho mundo virtual, a hacerse una mejor idea del avance que lleva en el juego e incluso a imaginarse cómo podría ser la vida en su propia aldea. También es de admirar la cantidad de buena documentación en español y los foros activos que hay, que ayudan al jugador novel a iniciarse en la complicada tarea de ser uno más en un mundo virtual lleno de entresijos, competencia, trucos y complicaciones sociales.

2.6.3. World Of Warcraft

Comúnmente conocido como *WoW*, es un juego de rol multijugador masivo en línea (*MMORPG*) donde miles de jugadores tendrán la oportunidad de aventurarse juntos en un mundo enorme, haciendo amigos, matando monstruos y participando en épicas búsquedas que pueden llevar días o incluso semanas. Lo asombroso de este juego es que consigue que más de diez millones de suscriptores paguen religiosamente una cuota mensual de 12 euros por jugar. También es asombroso el esfuerzo de producción que supone crear un mastodonte de las dimensiones de *World of Warcraft*, con varias razas, clases y profesiones que ofrecen experiencias de juego sustancialmente distintas y que requieren el diseño y creación de centenares de objetos, armas, hechizos, localizaciones, diálogos, misiones y mucho más. Debido a los increíbles beneficios que produce el *WoW* mundialmente, muchas otras compañías de videojuegos han creado mundos similares, los cuales es posible que mejoren la calidad de los gráficos e incluso de los contenidos, pero ninguno se acerca ni de lejos a la cuota de mercado conseguida por este título. Desde luego nadie puede negar que el *WoW* supone todo un hito



Figura 2.7: Logotipo del WoW

en la historia de los videojuegos.

Thearsmonsters es un *MMORPG* para navegador, sin gráficos animados ni 3D, sin embargo el *WoW* es un *MMORPG* basado en cliente, con sus gráficos y animaciones 3D. Teniendo en cuenta esta primera observación, parece que no hay nada en el *WoW* que pueda servir de ejemplo, y desde luego es imposible plantearse una producción similar para un proyecto fin de carrera, sin embargo, abstrayendo el apartado visual, se pueden imitar muchas buenas ideas con respecto al diseño conceptual.

Cuando *WoW* vio la luz, allá por finales del 2004, apenas tenía un pequeño subconjunto de todas las funcionalidades de las que dispone hoy en día. Al principio solo se podían hacer misiones, ir subiendo de nivel el personaje, vender productos en las subastas y poco más. Poco a poco se fueron añadiendo parches que incluían nuevas mazmorras, arenas *PvP*, nuevos eventos, nuevas misiones, nuevas cualidades para las profesiones, mejoras en las habilidades de algunas clases, etc. Sin embargo la primera versión ya era suficientemente completa y muy estable. Esa misma idea se va a aplicar en la elaboración de este juego, aprovechando el hecho de ser como una Web, donde es fácil realizar actualizaciones, y permite mantener entretenidos a los usuarios con nuevas y periódicas actualizaciones. Lo importante de esta idea es que lo que se vaya a incluir en las actualizaciones ya esté prediseñado de antemano, de forma que el modelo y el código implementado ya estén preparados para ser actualizados. Además, esta forma de elaborar el software en espiral permite mayor flexibilidad, y si por alguna razón el proyecto no triunfa y hay que abortarlo no se habrán malgastado esfuerzos en elaborar



Figura 2.8: Algunos personajes del *WoW*

cosas que al final no eran rentables.

Una propiedad muy importante de los juegos en línea es que permitan al jugador personalizar su personaje (o su guarida o su cuenta o lo que sea) para poder proyectar su personalidad en el mundo virtual que habitan. La expresividad de los personajes es muy buena en el *WoW*, mediante un menú de acciones, es posible saludar a otros personajes, bailar, insultar, eructar, pedir perdón, alabar, coquetear y un sinfín de acciones más. Aunque parezca una tontería no es raro encontrarse grupos de *orcos* y *trolls* bailando juntos en medio de una plaza de su ciudad principal. En *Thearsmonsters* no será posible incluir todas las características que se desearía, pero siempre que sea posible se permitirá que el jugador pueda expresarse, por ejemplo, eligiendo el nombre de cada monstruo que nace en su guarida, o escribiendo mensajes en las salas de los demás jugadores.

La interfaz del *WoW*, al igual que la de todos los títulos que produce *Blizzard* (que es la compañía que lo creó y lo mantiene en la actualidad), es impecablemente limpia e intuitiva. Todo lo que se puede hacer está presente en la pantalla a modo de iconos, que también se pueden configurar como teclas de acceso rápido, para permitir una mejor jugabilidad a nivel experto. Conseguir que la interfaz sea muy

sencilla es muy complicado. El hecho de que Blizzard haya empleado grandes esfuerzos para pulir la interfaz significa que es algo que no se puede dejar de lado.

El *WoW* también es conocido por lo suave que resulta la curva de aprendizaje del juego. Al comienzo los jugadores manejan unas pocas acciones disponibles y al final acaban controlando difíciles habilidades, relaciones sociales, mercado, tácticas de batalla, controles avanzados e incluso un eficaz trabajo en equipo, sin tener que leer ningún manual. Para lograrlo, se comienza la aventura con personajes que tienen muy pocas habilidades (dos o tres nada más) y se realizan misiones que más bien son para aprender a jugar. Se restringe el acceso a lugares que requieren conocimientos avanzados y no se permite la realización de ninguna profesión ni especialidad particular. Poco a poco, según avanza el juego, se van consiguiendo nuevas habilidades, que son cada vez más poderosas y difíciles de usar, se va dejando al personaje la posibilidad de especializarse en alguna profesión, van apareciendo misiones cada vez más complicadas que necesitan la elaboración de diferentes estrategias, se va permitiendo el acceso a lugares donde se realizan batallas contra otros jugadores y de esta forma se va abriendo un mundo de posibilidades. Con esto no solo se logra un aprendizaje más suave, sino que al desbloquear nuevas habilidades al completar misiones los jugadores se sienten gratificados por el esfuerzo, lo cual los mantiene mucho más entretenidos que si tuvieran todos esos poderes desde un comienzo. En *Thearsmonsters* no será diferente. Para empezar solamente habrá una raza disponible, con una única sala en la guarida. Para poder utilizar otras razas o construir otras salas habrá que ir completando misiones, de dificultad creciente. En otros juegos para navegador no hay misiones que completar, aunque también es habitual que el árbol de habilidades o tecnologías se vaya desbloqueando en base a unos requisitos. Por ejemplo en *Ogame*, donde al principio no se pueden hacer naves, pero a medida que avance el nivel de las minas se puede construir un laboratorio de tecnología, con éste se desarrolla la tecnología necesaria para construir una fábrica de naves, y con esta ya se pueden empezar a hacer naves pequeñas, para seguir consiguiendo cosas que permitan construir naves más grandes, y así sucesivamente hasta desbloquear



Figura 2.9: Captura de pantalla en uno de los múltiples escenarios del *WoW*

todas las tecnologías del juego.

La historia y el argumento del *WoW* son hechos basados en una saga más antigua llamada Warcraft. Al tratarse de un juego multijugador masivo la historia no puede convertir al personaje en protagonista de los hechos, como suele pasar en los juegos clásicos para un solo jugador. Realmente la historia que va sucediendo es la unión de pequeños sucesos que se producen en cada misión. Lo realmente importante es la ambientación, la sensación que se transmite al jugador de que realmente se está viviendo una aventura alternativa. Para lograr tal sensación, los diseñadores de *Blizzard* tratan de mantener la coherencia en cada detalle del mundo, nunca desvían la temática de una zona y absolutamente todo lo que se puede ver, oír y sentir está relacionado con el resto del ambiente, incluso la temática de las misiones tiene coherencia con el lugar donde se realizan, y esto no es algo exclusivo de *Blizzard* sino del diseño de videojuegos en general, solo que ellos son especialmente buenos en este aspecto. Los diseños en *Thearsmonsters* tienen especial cuidado por mantener la coherencia.

El comercio es otra parte fundamental del juego. Casi todo lo que consigue un personaje se puede intercambiar o vender. Para empezar, existe una subasta donde se pueden depositar objetos para que otros jugadores pujen por ellos. Así mismo también es un buen lugar para comprar cosas difíciles de encontrar.

Hay jugadores que hacen verdaderas fortunas teniendo un poco de cuidado con las tendencias de la subasta. Viendo el proceso con un poco de perspectiva, se podría decir que el lugar de donde provienen las materias primas es en el propio escenario donde se realizan las misiones. Generalmente el dinero, armas, escudos y demás se consiguen matando monstruos. Ciertas profesiones además pueden recolectar materiales especiales, por ejemplo los mineros pueden conseguir metales, los herboristas plantas y los pescadores peces, que después se puede vender en la subasta. Luego hay otras profesiones que pueden manipular materias primas para conseguir otras más específicas, por ejemplo un alquimista puede mezclar distintos tipos de planta para crear pócimas y los herreros trabajan los metales conseguidos para crear armaduras. Habiendo muchas profesiones disponibles, cada personaje solo puede especializarse en dos, debiendo comprar en la subasta todo aquello que no pueda fabricar por sí mismo. Dejar libertad de comercio hace que, gracias a la ley de la oferta y la demanda, los precios se regulen automáticamente. Con este factor aparecen muchas alternativas de juego. De hecho en el *WoW* casi siempre se pueden conseguir las cosas de varias formas distintas, y además siempre hay muchas cosas que hacer. Lo normal es tener varias misiones activadas e ir completándolas en el orden que más agrade al jugador, y no es necesario completarlas todas para avanzar. Por ejemplo, se puede subir de nivel realizando todas las misiones en una zona de forma individual o bien participando exclusivamente en las más difíciles con la ayuda de un equipo de compañeros. Este concepto es aplicable directamente a la economía de *Thearsmonsters*: los jugadores se verán beneficiados económicamente si especializan su guarida en unas pocas profesiones para ofrecer los mejores servicios en la zona a la que pertenezcan. Además podrán conseguir dinero bien a través del comercio o bien realizando misiones especiales de lucha.

A pesar de todo lo que se puede hacer en el *WoW*, la parte principal de el juego son las batallas, ya que todo lo que se consigue para el personaje está enfocado a adquirir mayor nivel para ser más poderoso. Pero pronto se descubre que el poder que tenga un jugador no lo es todo, normalmente es necesario pensar

las cosas antes de hacerlas. Por ejemplo, para poder matar al demonio jefe de un castillo es necesario planear un asalto entre varios compañeros, utilizando de forma provechosa las diferentes habilidades de las que dispone cada uno. El trabajo en equipo y el juego organizado es vital, y aquí es donde entra otro aspecto importantísimo: la colaboración. El juego está diseñado de tal manera que siempre es mejor un grupo combinado y coordinado de varias razas y clases que otro con todos los personajes del mismo tipo, y resulta que la experiencia de hacer un buen trabajo con un grupo de amigos es mucho más divertido y satisfactorio que ganar sólo por fuerza bruta. Nuevamente se extrapolará este concepto a *Thearsmonsters*, donde cada raza tendrá unas habilidades distintas que se podrán combinar con otras, y cuando se puedan hacer grupos formados por monstruos de jugadores distintos, el resultado será más especializado y poderoso. Aunque evidentemente las posibilidades serán mucho menores que en el *WoW*.

En resumen, se analiza el *World of Warcraft* porque es un ejemplo difícilmente superable de cómo se debe hacer asequible un juego que, a priori, es tremadamente complejo. El secreto de *WoW* es su tremenda sencillez y accesibilidad, así como su perfecta curva de aprendizaje y dificultad. Cualquier novato, cualquiera, puede empezar a jugar sin problemas y verse enganchado en esa mecánica que hace que cada vez todo resulte un poco más complicado y a la vez más divertido. El perfil de jugador de *World of Warcraft* es muy curioso, porque es teóricamente un juego para *hardcore gamers*, debido a la ingente cantidad de horas que exige, y sin embargo está lleno de personas que no suelen ser jugadores asiduos. Es un juego para todo el mundo, que gusta a los más exigentes y a los más casuales, aunque es difícil librarse de la dinámica que arrastra a los jugadores a no poder dejar el juego cuando tienen otras cosas más importantes que hacer en la vida real.

Capítulo 3

Concepto y Análisis

Índice general

3.1.	Resumen del juego	36
3.2.	Sistema de juego	39
3.2.1.	Tareas programables	39
3.2.2.	Economía	41
3.2.3.	Misiones	43
3.2.4.	Ciclo generacional	44
3.3.	Modelo conceptual	45
3.4.	Usuarios	46
3.4.1.	Jugadores	46
3.4.2.	Administradores	51
3.5.	Salas de la guarida	52
3.5.1.	Como se usan	53
3.5.2.	Realizar obras en una sala	54
3.5.3.	Publicación	57
3.5.4.	Restricciones de acceso	58
3.5.5.	Tipos de sala disponibles	59

3.6. Monstruos	61
3.6.1. Estado de un monstruo	61
3.6.2. Ciclo de vida de los monstruos	67
3.6.3. Monstruos en acción	69
3.7. Misiones	70
3.7.1. Realización de asaltos en una misión	70
3.7.2. Templos y salas con PNJs	71
3.7.3. Sistema para desbloquear razas y salas	72
3.8. Batallas	73
3.8.1. Influencia de las habilidades de batalla	74
3.8.2. Inicialización de la batalla	74
3.8.3. Inicialización de las rondas	74
3.8.4. Factores de cada turno	76
3.8.5. Finalización de la batalla	76

EN este capítulo se analizan los aspectos relevantes del juego, se aclaran términos y se fijan las restricciones del modelo conceptual. Se introduce el vocabulario que describe el dominio, como los monstruos, las guaridas, las salas, las misiones, atributos primarios de un monstruo, etc. De igual forma se irán comentando las decisiones tomadas y las ideas descartadas que fueron dando forma al juego durante las fases de diseño conceptual y de análisis.

3.1. Resumen del juego

Thearsmonsters es un juego en línea multijugador masivo (MMORPG) basado en navegador. Para jugar solamente es necesario tener un navegador web compatible con acceso a Internet, entrar en la URL de la página principal y crearse una cuenta. No es necesario instalar ni descargar software adicional.

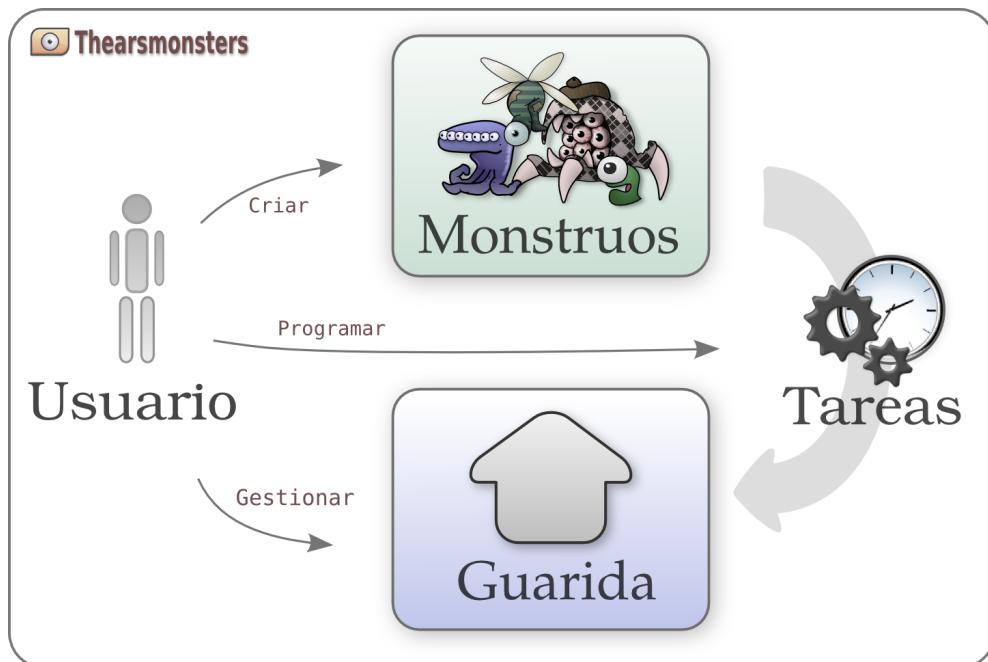


Figura 3.1: Esquema básico del sistema de juego

El esquema de la figura 3.1 es la visión más genérica que se puede tener del modelo y de los casos de uso en los que se basa el juego. Un usuario registrado controla su entorno criando nuevos monstruos, programando sus tareas y gestionando su guarida. Los monstruos nacen y crecen, mejoran sus habilidades para efectuar tareas y finalmente mueren. Las tareas se realizan en las salas de la guarida, que están especializadas para cada uso concreto. Parte de las tareas consiste en construir nuevas salas y mejorar las instalaciones, de forma que aunque los monstruos van pasando por varias generaciones, la guarida se mantiene y solamente puede mejorar.

La historia del juego dice que de vez en cuando, en algún lugar del subsuelo, nace un nuevo monstruo-planta llamado el *Ojo de la vida*, que cava una pequeña guarida a su alrededor, preparando la aparición de una nueva familia de monstruos. Este es el comienzo del juego. Primero deberán nacer las primeras crías del Ojo de la vida, que tendrán que completar unas sencillas misiones para poder adquirir

el conocimiento necesario para construir nuevos tipos de salas. Una vez completadas se podrá empezar la recolección de basura, y así ir ampliando y mejorando las salas creadas. Lo siguiente será desbloquear nuevas razas de monstruo, mucho más fuertes y longevas que las primeras, y con estas nuevas criaturas se podrán completar misiones más difíciles, con mejores premios. El objetivo del Ojo de la vida será incubar y cuidar de las nuevas crías, que se pueden conseguir bien a través del mercado de huevos o bien a través de la reproducción entre monstruos.

Para mejorar la gestión de la guarida será necesario ir especializando cada monstruo en una profesión, de forma que se aproveche al máximo la colaboración entre ellos. Una vez alcanzado cierto nivel en el transcurso del juego, comenzará la interacción económica entre diferentes guardias, que pertenecen a diferentes jugadores. Algunas salas pueden ser publicadas, permitiendo el acceso a monstruos de otras guardias a cambio de un precio regulable. Así mismo, en algunos casos será más rentable pagar por el servicio ofrecido en la guarida de otro jugador que construir y mantener todas las salas necesarias para obtener el mismo servicio de forma autónoma.

Los jugadores también podrán juntarse en grupos llamados *alianzas*. Para realizar misiones multijugador, mucho más difíciles que las habituales, se podrán unir varios monstruos dentro de la misma alianza, no necesariamente del mismo jugador, y así obtener premios aun mayores, que habrá que repartir entre los ganadores. También se podrá participar en arenas y torneos entre diferentes alianzas, es decir luchas *PvP* (jugador contra jugador), que serán el objetivo de los jugadores expertos.

Una de las características principales de este juego es que no se puede atacar la guarida de otro jugador, dejando tiempo libre para desconectarse del juego. Sin embargo será necesario prestarle atención de forma regular para controlar los cambios que se vayan produciendo y así maximizar la producción. El avance en el juego se produce al realizar misiones, y éstas se realizan cuando el jugador lo deseé. No se exigen demasiadas obligaciones al jugador. Este punto de vista tratará de permitir la participación de jugadores casuales, que no tienen cabida en la mayoría

de juegos que hay hoy en día en Internet. Los jugadores exhaustivos (*hardcore players*) también podrán competir por los primeros puestos en los *rankings* y luchar en batallas PvP.

3.2. Sistema de juego

El juego está basado en las reglas que impone el modelo conceptual, sin embargo también está muy condicionado por el medio físico en el que se implementa, es decir, en un servidor Web Java EE. En primer lugar, no puede ser demasiado complicado manejar el juego desde un navegador Web. También hay que tener en cuenta que, al contrario de otros estilos de juego, el usuario no puede estar siempre conectado, sin embargo el mundo virtual en el que está debe seguir funcionando. En resumen, hay ciertas características del modelo que han sido diseñadas solamente para facilitar el control del juego.

3.2.1. Tareas programables

Las tareas permiten organizar y controlar lo que hacen los monstruos durante las 24 horas del día sin necesidad de tener que conectarse repetidamente al juego. Los monstruos están continuamente realizando tareas como recolectar basura, construir, ampliar la guarida, limpiar una sala, cocinar, enseñar habilidades, etc. El jugador podrá de esta forma ordenar a los monstruos qué tarea deben hacer en cada momento para controlar el progreso del juego. Lo malo es que esta solución añade mucha complejidad al sistema informático.

Para permitir una automatización indefinida, las tareas se van a organizar en días, de forma que lo que se programa son las tareas que se realizan en un día del juego, incluyendo dormir y comer. Si no hay cambios en la lista de tareas entonces se repiten las mismas tareas cada día. Además las tareas diarias se dividen en franjas horarias, que facilitan la organización de las tareas tanto desde el punto de vista conceptual como informático. Así un día se pueda dividir en 24 franjas horarias por ejemplo, y en cada una de ellas se fija una tarea. Si se desea que una

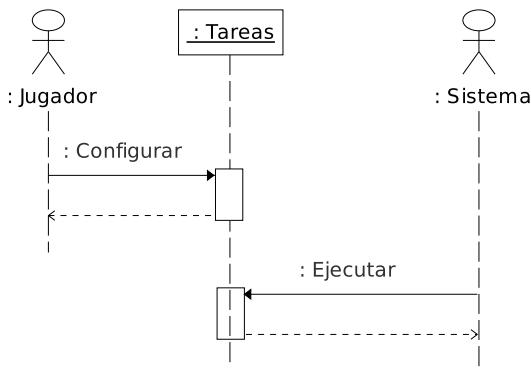


Figura 3.2: El usuario asigna tareas a un monstruo y el sistema las realiza de manera automática.

tarea dure más de una franja horaria entonces basta con repetir la misma tarea en varias franjas horarias.

Por cuestiones de jugabilidad, cuando se quiera asignar una nueva tarea, el sistema debe mostrar solo aquellas tareas que sean posibles para ese momento. Por ejemplo, si se desea llevar un monstruo al gimnasio a cierta hora y ya estaba programado que a esa hora el gimnasio iba a estar completo, entonces no se debería dar la opción de realizar esa tarea o bien avisar de que no es posible. Dicho de otra forma, cuando el usuario esté organizando las tareas de un monstruo debe saber en todo momento lo que puede realizar, lo que no puede realizar y el porqué. Para llevar el control de la cantidad de monstruos que habrá en una sala durante cada franja horaria, la base de datos debe estar preparada y optimizada para almacenar todas las tareas de cada monstruo y devolver la información necesaria para crear los desplegables que utiliza el jugador. Otro aspecto que se debe controlar es si el monstruo en cuestión, aunque haya plazas libres, puede o no entrar en dicha sala. Por ejemplo, en la guardería solo pueden entrar crías de monstruo, no adultos ni ancianos. Este aspecto realmente no está relacionado con el hecho de que las tareas se dividan en franjas horarias, pero está ligado a las opciones disponibles que se mostrarán en la interfaz.

Con este sistema, el juego entero se controla básicamente asignando tareas a los monstruos. En diagrama de la figura 3.2 resume el concepto; los jugadores

configuran el estado de las tareas (en cualquier momento) y tras cada franja horaria el sistema se encarga de que se ejecuten automáticamente.

3.2.2. Economía

El modelo económico del juego facilita la mayoría de las relaciones entre jugadores, ya que aunque no será posible atacar por la fuerza la guarida de otro jugador, sí que será posible entrar pacíficamente en ella y utilizar alguno de sus servicios a cambio de un módico precio.

Por ejemplo, en un gimnasio se mejora la fuerza base de un monstruo, por lo tanto si se quiere mejorar la fuerza de algún monstruo no hay más que indicarle en su lista de tareas que visite de vez en cuando el gimnasio. Este gimnasio puede construirse y mantenerse en la propia guarida del monstruo, sin embargo, para poder gozar de un gimnasio grande y de calidad será necesario invertir mucho esfuerzo en construir, ampliar y actualizar el mismo. Tal vez el jugador necesite aumentar la fuerza de un monstruo y no disponga del tiempo necesario para crearse su propio gimnasio, en ese caso lo que puede hacer es utilizar el gimnasio de otro jugador.

El precio de la entrada por cada turno lo fija el dueño de la sala. El dueño debe tener en cuenta a la competencia para encontrar el precio de máxima rentabilidad, un precio demasiado elevado implica que la sala no será muy visitada, en cambio un precio demasiado reducido no producirá todos los beneficios que podría. Además también hay que tener en cuenta el tamaño de la sala, ya que el tamaño determina el número de monstruos que pueden utilizarla de forma simultánea. Evidentemente, cuantos más monstruos puedan entrar, mayores serán los beneficios, pero una sala demasiado grande es muy costosa. En general, cuanto mejor sea el servicio, mayor precio se podrá cobrar por él. Un buen jugador se preocupará de ofrecer los servicios que tengan mayor demanda en su zona geográfica.

Como puede verse en la figura 3.3, en este juego hay dos tipos de recursos: la basura y el dinero. Los monstruos pueden recolectar basura como parte de sus tareas diarias y almacenarla en el almacén de basura. Después se puede intercam-

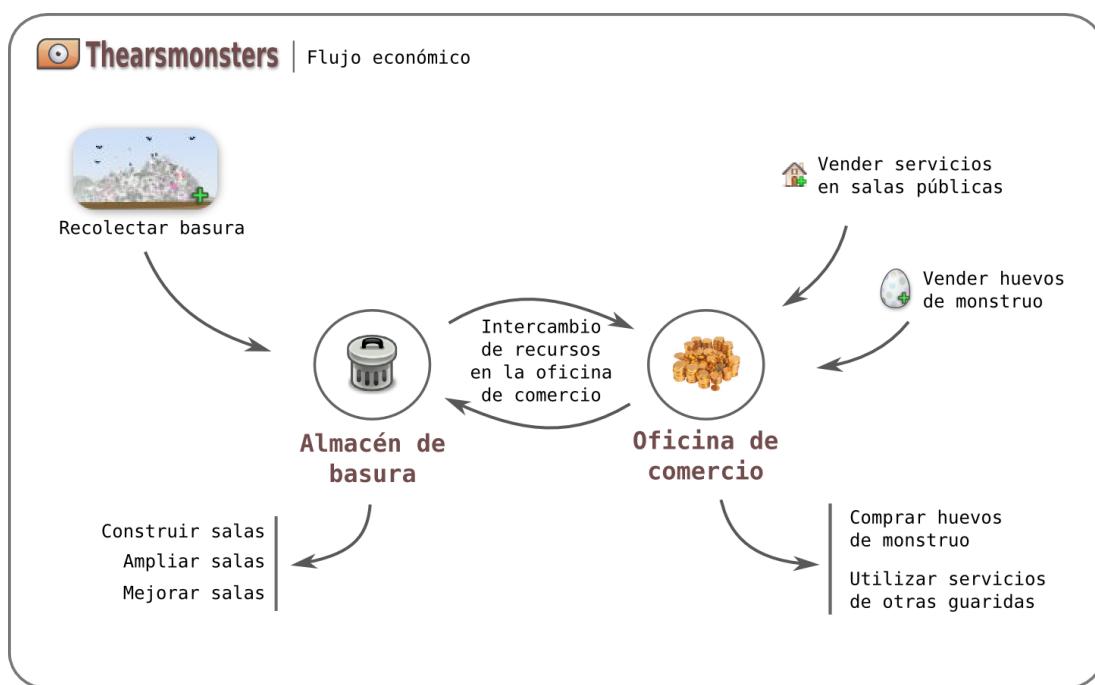


Figura 3.3: Flujo de los recursos económicos. Dónde se obtienen, almacenan y utilizan en el juego.

biar basura por dinero utilizando la oficina de comercio. La cantidad máxima de basura y dinero que se pueden almacenar dependen del tamaño de cada una de las salas anteriores respectivamente. La basura sirve para mejorar la guarida y el dinero para comerciar con la máquina o con otros jugadores. Los huevos de monstruo pueden o bien comprarse o bien pueden producirlos los propios monstruos. En caso de exceso éstos podrán venderse y así también producir beneficios (el precio de venta es la mitad del precio de compra).

En general, la recolección de basura es el recurso principal de los monstruos, y es un bien igualmente accesible para todos los jugadores. El flujo económico va desde el basurero del exterior hasta las guaridas de los jugadores y después tiende hacia los jugadores que ofrezcan los mejores servicios.

3.2.3. Misiones

Los monstruos, además de trabajar en las tareas diarias de la guarida, podrán participar en batallas que les ayuden a conseguir diferentes objetivos. Las diferentes razas de monstruo que se pueden criar en la guarida así como los diferentes tipos de sala que se pueden construir deberán ser desbloqueados completando las misiones que hay en los templos. Las misiones generalmente no son más que luchar contra otro grupo de monstruos manejados por la máquina hasta acceder a la sala final de cada templo, donde suele haber otro monstruo final más fuerte, que hay que eliminar para conseguir el premio principal del templo. Así que para avanzar en el juego será necesario contar con un buen equipo de combate, el cual se consigue gracias a buen sistema económico. Los monstruos se pueden especializar para la lucha, siendo tarea del jugador el decidir qué cantidad de monstruos se dedicarán al mantenimiento de la guarida y qué cantidad a perfeccionar sus habilidades de combate.

En la sección 3.7 se explica en más detalle la estructura, funcionamiento y desarrollo de las misiones.

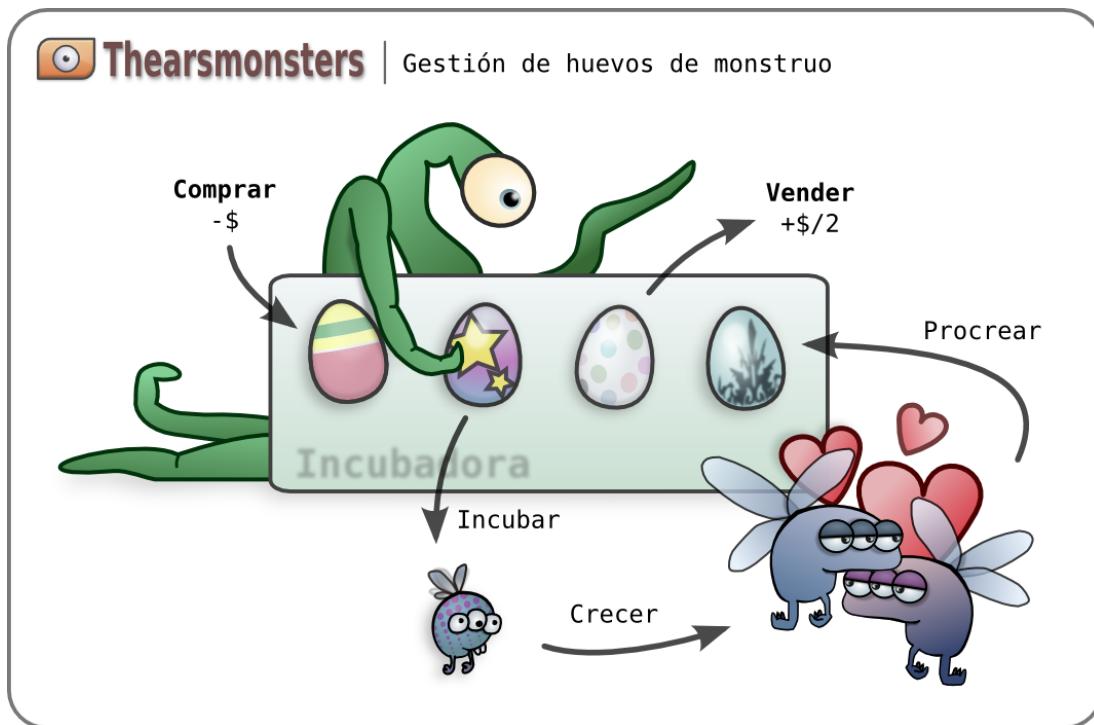


Figura 3.4: Los huevos pueden costar dinero (dependiendo de la raza de monstruo son más caros o más baratos), o también pueden suponer una fuente de ingresos, si el jugador cuenta con los medios necesarios para producirlos.

3.2.4. Ciclo generacional

Un aspecto muy interesante del estilo de juego en *Thearsmonsters* es la manera de conseguir nuevos monstruos. En las guaridas hay una cantidad limitada de *espacio vital*, determinada por el tamaño de los *apartamentos* (en el apartado 3.5.5 se enumeran todas las salas que hay disponibles y su función principal), que condiciona el número de monstruos que se pueden tener de forma simultánea. En el apartado 3.6.2 se comentan los estados por los que va pasando un monstruo, desde que sale del huevo hasta que muere. Cuando un monstruo se muere, libera el espacio vital que ocupaba y entonces el jugador puede incubar otro huevo para dar paso a una nueva criatura en la guarida.

Como se puede apreciar en la figura 3.4, la incubadora del *ojo de la vida*

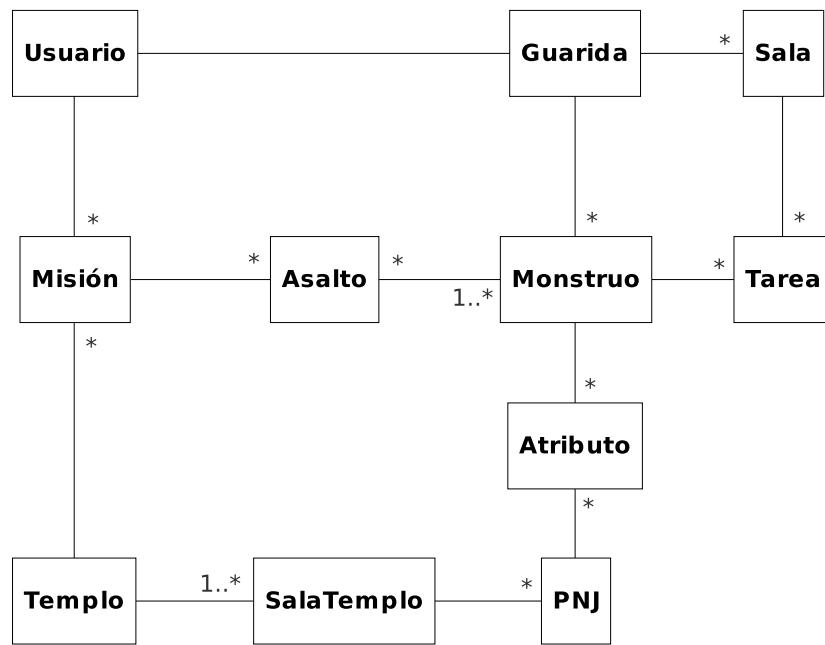


Figura 3.5: Una primera aproximación al modelo conceptual del dominio.

puede albergar cierta cantidad de huevos, los cuales se pueden obtener mediante la reproducción de los monstruos adultos de la guarida o simplemente comprándolos. A continuación éstos se pueden vender (a la mitad de precio que vale comprarlos) o incubar (si hay suficiente *espacio vital* libre en la guarida) para obtener nuevas crías de monstruo.

3.3. Modelo conceptual

El diagrama de la figura 3.5 representa a grandes rasgos las clases del dominio que se pueden encontrar en la aplicación.

Vemos que cada jugador tiene una única guarida que está compuesta de salas y que tiene varios monstruos. Los monstruos tienen una serie de atributos que determinan su fuerza, agilidad, inteligencia y demás habilidades. Las tareas organizan las actividades que realizan los monstruos en las salas durante cada turno.

Por otro lado están las misiones. Cada jugador puede realizar tantas misiones como tenga desbloqueadas. Cada misión se realiza en un templo que está formado por salas. Cada sala tiene *PNJs* (monstruos controlados por la máquina) que hay que eliminar para pasar de sala. Cada jugador solo puede realizar una misión por cada templo desbloqueado. Sobre esa misión puede organizar tantos asaltos como desee, aunque solamente podrá hacerse un asalto por franja horaria. En cada asalto se intentará eliminar a los *PNJs* de la sala correspondiente, si se consigue, el próximo asalto se hará sobre la próxima sala (están organizadas secuencialmente) o bien se completa el templo (si era la última sala). Al contrario que las tareas (que se repiten diariamente) los asaltos solamente se realizan una vez. Las misiones almacenan la última sala completada del templo para saber a qué sala deberá ir el próximo asalto.

En sucesivos capítulos se irá ampliando el detalle.

3.4. Usuarios

En esta sección se analizarán los diferentes usuarios del juego, su representación dentro del mismo y su descripción en el mundo real. El sistema interactuará con jugadores, administradores y otros sistemas. Los jugadores pueden tener gustos y preferencias diferentes. Los administradores serán los encargados de mantener las sociedades que se produzcan dentro del juego y de asegurar que se cumplan las leyes establecidas, es decir, que tendrán el derecho a expulsar jugadores si hacen trampas. También se puede considerar otro usuario al sistema de invocación de eventos, que tendrá que activar la actualización del *cron* cada vez que transcurra una franja horaria dentro del juego.

3.4.1. Jugadores

Los jugadores son los que controlan los monstruos de la guarida. Como puede verse en la figura 3.6, cada jugador posee una única guarida. Aunque la relación sea 1 a 1, se separan las entidades por cuestiones conceptuales. Con el jugador

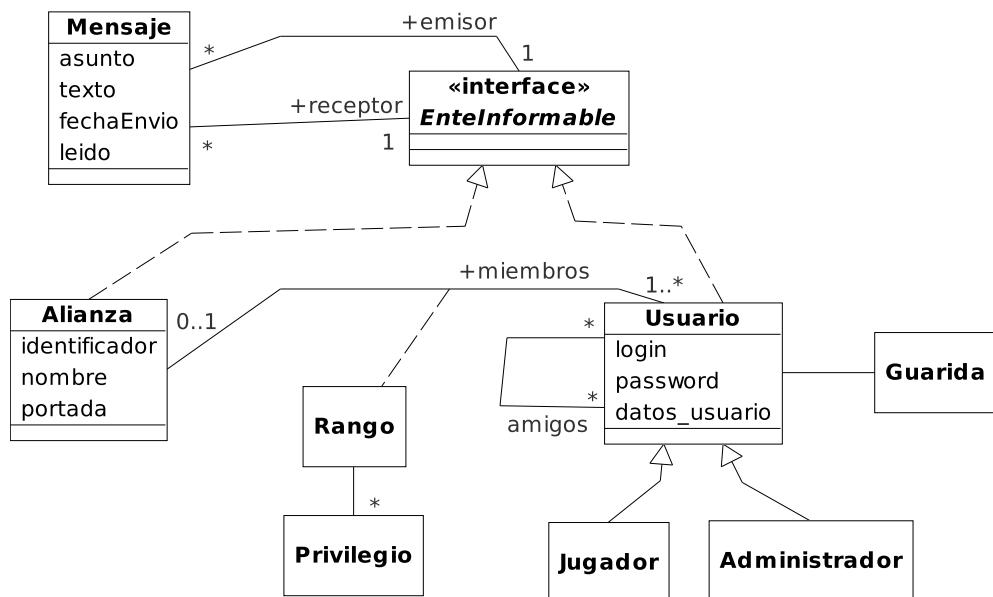


Figura 3.6: Usuarios del juego y su organización

irán todos los atributos relacionados con los usuarios (contraseña, puntuación en los *rankings*, datos personales, etc.) y con la guarida los que sean relevantes dentro del juego (cantidad de recursos, posición geográfica, salas desbloqueadas, etc.)

3.4.1.1. Perfil de jugador

Desde el punto de vista del *Marketing*, se puede considerar que el juego es apto para cualquier tipo de jugador, de todas las edades a partir de 10-14 años, que nunca hayan jugado a un juego de estas características o bien que estén aburridos de jugar siempre a lo mismo con juegos para navegador típicos. Se sigue un poco el perfil de internauta pasando ratos libres delante del ordenador, de hecho el argumento del juego tiene en cuenta un colectivo de internautas que conocen la religión del *pastafarismo*, que se ha extendido por muchos *blogs* y *foros* de Internet.

Este juego pretende ser un entretenimiento adicional a los jugadores casuales que utilizan Internet habitualmente pero que no tienen tiempo suficiente para

estar todo el rato pendientes del juego. No está enfocado a jugadores compulsivos (los llamados *hardcore players*) que puedan estar conectados todo el rato para adueñarse del mundo virtual en el que juegan y abusar de los novatos porque simplemente no podrán hacerlo, sin embargo, seguramente haya jugadores que encontrarán la forma de ser mejores que los demás, ya sea mediante superioridad económica o táctica, aunque al menos no podrán molestarán al resto.

En la sección 2.5 del capítulo Contextualización se hace un análisis de los diferentes tipos de jugadores que hay.

3.4.1.2. Organización de los jugadores: Alianzas y Amigos

Como indica la figura 3.6, los jugadores tienen la posibilidad de agruparse en alianzas. Los miembros de la misma alianza podrán hacer grupos de monstruos conjuntos para realizar misiones multijugador o participar en torneos entre alianzas. Los casos de uso que puede realizar un miembro de una alianza se muestran en la figura 3.7

El sistema de alianzas es muy similar al que utiliza el juego *Ogame* [15] porque es muy flexible, permite una administración distribuida de la alianza distribuida entre los miembros de la misma y hace que los jugadores se sientan importantes cuando adquieren un nuevo rango con más privilegios.

Cada jugador también tiene una lista de amigos, donde puede ver los que están actualmente en línea además de tener un enlace rápido en cada uno para ver sus datos, escribir mensajes o iniciar un *chat*. Inicialmente la lista es vacía, aunque en cuanto se pertenezca a una alianza la lista se llena con todos sus miembros. Para ampliar la lista se envían solicitudes a otros jugadores, que si las aceptan se añade cada uno de los jugadores en la lista del otro, es decir que aceptando solicitudes de otros jugadores también se amplía la lista. La relación de amistad es simétrica, no puede haber un jugador que tenga a otro como amigo y esto no se cumpla en sentido contrario.

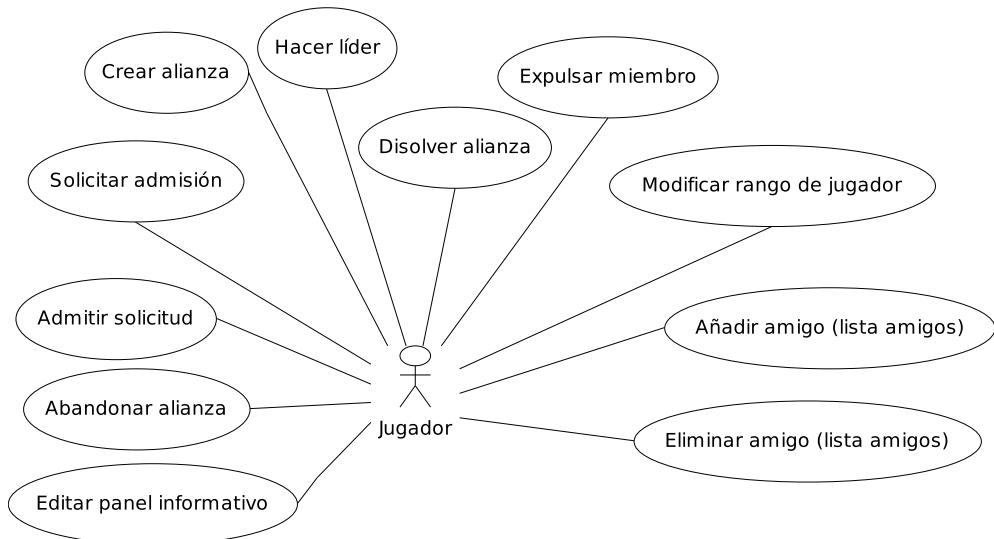


Figura 3.7: Casos de Uso de un miembro de una alianza. Algunos de ellos solo se pueden realizar si se dispone de los privilegios adecuados.

3.4.1.3. Comunicación entre jugadores: Mensajes

La comunicación entre jugadores es fundamental en un juego multijugador, sin ella sería imposible causar la sensación de que hay alguien más en ese mismo mundo virtual. En la mayoría de *MMORPGs* los jugadores pasan tanto tiempo jugando como charlando con los demás jugadores. En un *MMORPGs* para navegador, la comunicación puede y suele realizarse a través de otros medios externos como *foros* o *Messenger*, pero si se integra en el sistema se facilita mucho la jugabilidad a usuarios poco expertos en el uso de Internet, cosechando un número mayor de jugadores.

Como se ve en la figura 3.8, en este juego los usuarios podrán enviarse mensajes entre sí para mantenerse comunicados (también pueden enviarse mensajes entre alianzas). Para enviar un mensaje a todos los miembros de una alianza basta con poner de destinatario el nombre de esa alianza (la alianza recibe el mensaje y lo distribuye automáticamente), aunque solo se podrán enviar mensajes a la propia alianza a la que pertenece el emisor, a menos que se disponga del privilegio de

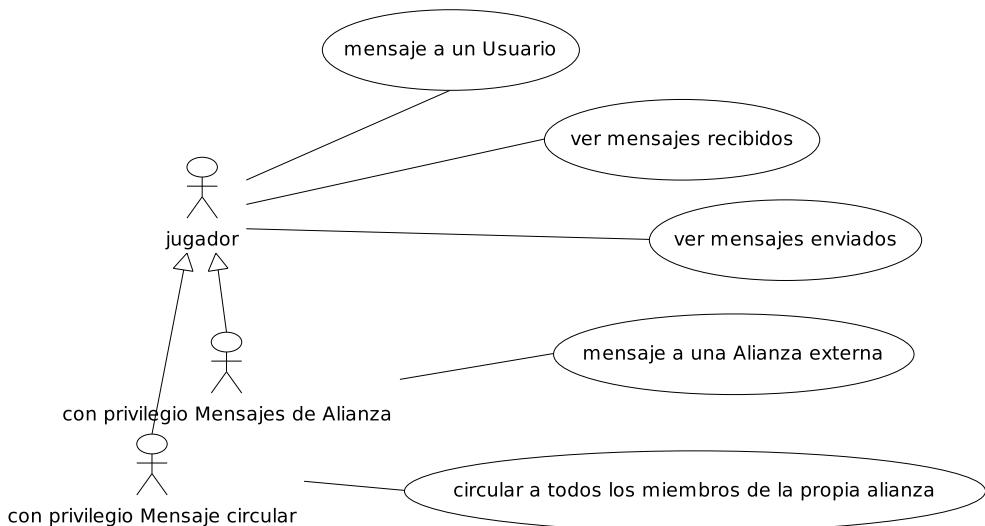


Figura 3.8: Los mensajes que puede enviar o recibir un jugador también dependen de los privilegios dentro de su alianza.

mensajes de alianza (además los mensajes externos dirigidos a la alianza llegan solamente a los miembros con este privilegio).

Cada jugador tiene dos bandejas de entrada: la que recibe mensajes de otros jugadores o alianzas y la que recibe mensajes del sistema, como reportes de batalla o información de las misiones (los mensajes del sistema se generan automáticamente al suceder los eventos).

3.4.1.4. Competencia entre jugadores

Los jugadores competitivos necesitan poder compararse con los demás para tener motivación. Para ello se permitirá visualizar cierta información de la guarida de los demás, navegar entre las diferentes guardias a través de un mapa geográfico, tener siempre disponible información de los compañeros de alianza y lo que es más importante, un sistema para contabilizar puntos que permita situar a los jugadores en una lista.

Para poder situar a un jugador con respecto a los demás, se crea un *ranking* por cada objeto puntuable que sea posible comparar. Para que las puntuaciones

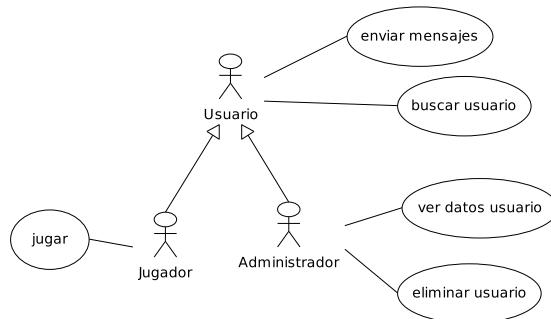


Figura 3.9: Casos de uso diferenciando a los jugadores normales de los administradores.

estén equilibradas, se multiplica el resultado que contabiliza los puntos por un factor que pondera su valor (por ejemplo para los puntos de un jugador se suman los puntos de su guarida y de sus monstruos, y hay que definir la importancia de cada una de esas dos cosas utilizando los pesos).

Estos *rankings* serán accesibles por un enlace del menú de juego, y en la página que los muestre habrá diferentes pestañas para organizarlos.

3.4.2. Administradores

Será necesario disponer de un perfil especial para los administradores, que tendrán acceso a una sección especial donde puedan controlar la partida que se está jugando en el servidor donde se encuentran (panel de administración). Al igual que los demás usuarios dispondrán de una dirección de correo, donde los jugadores pueden escribir para denunciar ilegalidades.

En el panel de administración se podrán buscar jugadores en una caja de texto donde se introduce un nombre y se buscan resultados similares. La lista de jugadores que se muestra contiene información del jugador (login, nombre, puntuación, etc) y una serie de enlaces que permiten realizar diferentes acciones: ver mensajes (accede a la página donde se muestran los mensajes enviados y recibidos por el jugador), ver más información (muestra todos los datos posibles del jugador), expulsar jugador (pide confirmación, y si se acepta, se borra al

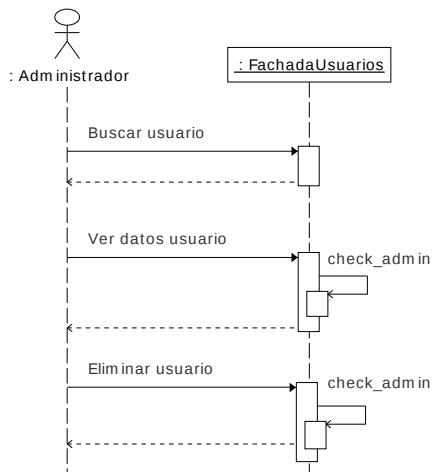


Figura 3.10: Las acciones que son exclusivas de un administrador son comprobadas por el sistema.

jugador expulsado de la base de datos) y enviar mensaje a jugador (para avisarlo si está realizando alguna actividad sospechosa).

También tendrá un buscador de salas públicas como el que tendrán los jugadores para buscar salas con las características deseadas.

En principio estas serán las funcionalidades de las que disponga el administrador, pero en el futuro, a medida que se vayan descubriendo nuevos tipos de trampas realizadas por los jugadores, se podrán añadir herramientas que faciliten su detección y posible solución.

3.5. Salas de la guarida

La guarida del jugador está dividida en salas, que son construidas y mantenidas por sus monstruos. En la figura 3.11 se indica la estructura de las salas. En una guarida hay varias salas de tipos distintos. Cada tipo de sala tiene un fin específico y sirve para mejorar alguna característica de los monstruos que la utilizan. Las salas tienen un nivel y un tamaño (medido en plazas). Hay que actualizarlas para mejorar el nivel (así mejoran el servicio ofrecido), y ampliar

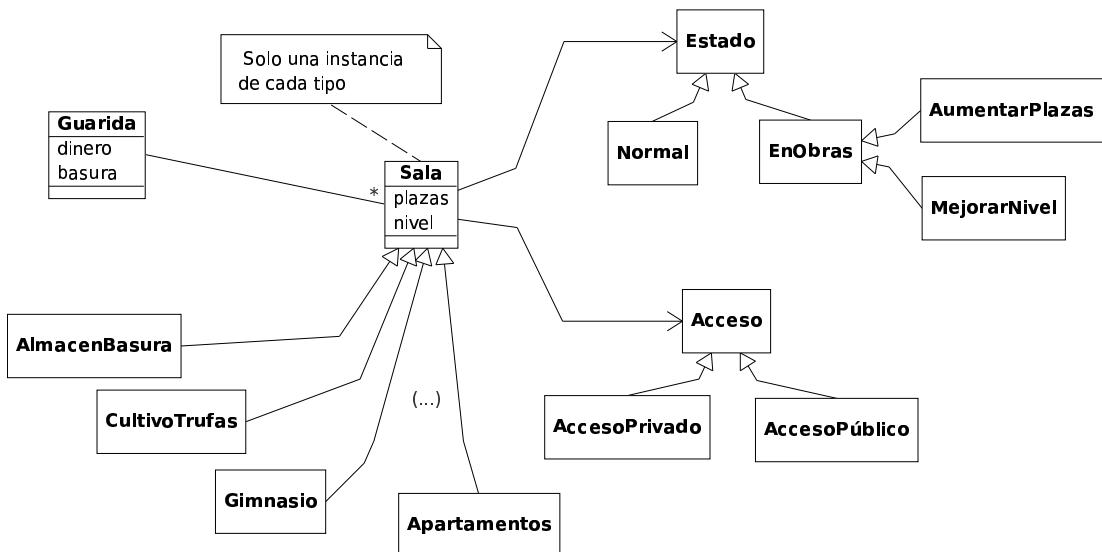


Figura 3.11: Estructura de las salas de la guarida.

su tamaño para dar servicio a un número mayor de criaturas simultáneas. Se puede alquilar el servicio de una sala a otros jugadores, cobrando por la entrada y así obteniendo beneficio económico (salas con acceso público). La calidad de una guarida, y por consecuente del jugador que la posee, depende del nivel de las salas que hay en su interior.

3.5.1. Como se usan

En la figura 3.12 se indican las acciones que puede realizar un jugador sobre la guarida y sus salas. Para que un monstruo utilice una sala hay que ir a su lista de tareas y si en esa sala se permite algún tipo de tarea accesible por ese monstruo, ésta saldrá en las opciones para asignar una nueva tarea. Si se elige una tarea en una sala, queda grabada y cuando sea el momento oportuno ya se llevan a cabo las acciones pertinentes automáticamente. Por ejemplo, se puede seleccionar un monstruo y decirle que entre en el gimnasio como instructor, después seleccionar a otros para que entren como usuarios (durante la misma franja horaria) y cuando llegue el momento mejorará su atributo de fuerza mientras estén dentro, tanto

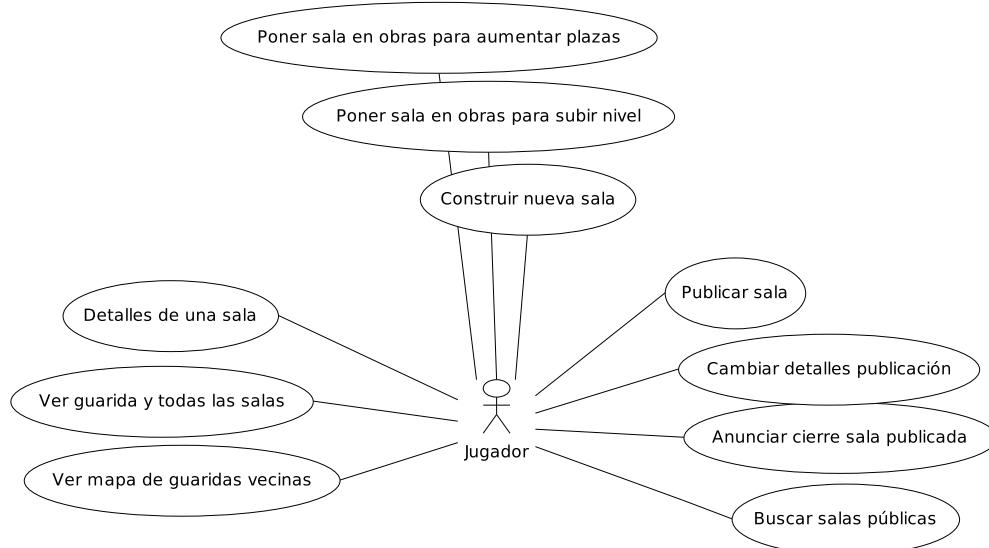


Figura 3.12: Casos de uso del jugador relacionados con la guarida.

como bueno sea el instructor y mejor sea el nivel de la sala, de forma automática.

3.5.2. Realizar obras en una sala

Como se indica en el diagrama de estados de la figura 3.13, las salas pueden estar en estado normal (cuando funcionan normalmente) o en obras (cuando se está ampliando o actualizando). El recurso necesario para construir o mejorar salas es la basura recolectada del exterior, además del tiempo y esfuerzo que les supone a los monstruos obrero cavar el hueco necesario y colocar los materiales en su sitio.

Las obras se pueden cancelar para devolver a la sala a su estado normal y entonces se devuelve al jugador la basura gastada, pero la siguiente vez que se vuelva a iniciar la obra hay que empezar desde el principio (el tiempo y esfuerzo realizados no se pueden recuperar). El estado en obras puede afectar al rendimiento de la sala (se especifican los efectos de las obras en cada tipo de sala) ya que se supone que hay ruido y escombros durante el avance de la misma.

Desde la vista de cada sala hay enlaces para ampliarla o actualizarla y en cuan-

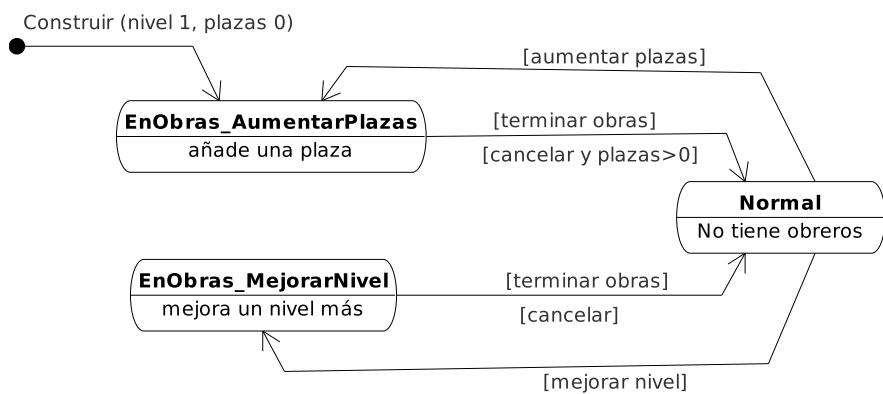


Figura 3.13: Diagrama de estados de una sala de la guarida (puede variar ligeramente dependiendo del tipo concreto de sala, pero casi siempre se mantiene este esquema).

to se pulsen se pone la sala en obras. Una vez iniciada la obra se pueden añadir monstruos ordenándoles ir a esa sala en su lista de tareas. Cuantos más monstruos haya simultáneamente en la obra, antes se terminará. Una de los atributos que tienen los monstruos es el de construcción, que cuanto mejor sea más rápido hará que avance la obra. Por decirlo de otro modo, al iniciar una obra, aparte de consumir la basura necesaria, se establece el esfuerzo que hay que hacer para realizarla. Cada monstruo, dependiendo de su atributo de construcción, realiza cierto esfuerzo por cada turno. Lo que se avanza en la obra durante cada turno es igual a la suma del esfuerzo realizado por cada uno de los obreros que haya en su interior. Cuando se complete todo el esfuerzo necesario se vuelve a poner la sala en estado funcional, y no se permite el acceso a monstruos haciendo de obreros. Además se cambia automáticamente la tarea de construcción de cada monstruo implicado en la obra por la tarea estándar de la guarida (sino intentarían entrar como obreros, no se permitiría su acceso y perderían las franjas horarias correspondientes sin hacer nada). Todo esto también es para que el juego pueda seguir sin que el jugador tenga que estar constantemente conectado.

3.5.2.1. Crear salas nuevas

En la vista de la guarida, si hay nuevos tipos de sala desbloqueadas (disponibles por haber completado las misiones correspondientes) y si se dispone de la basura necesaria, se puede iniciar la construcción de una sala nueva. Entonces se descuenta la basura necesaria inmediatamente (o un mensaje de error si no hay basura suficiente), y aparece una sala nueva de nivel uno, con cero plazas de tamaño, en obras para ampliar tamaño (y así cuando termine la obra la sala será de nivel uno con una plaza de tamaño). El tipo de sala que se ha empezado a construir desaparece de la lista de nuevas salas disponibles porque solo puede haber una sala de cada tipo dentro de la guarida del jugador. Este proceso se repetirá pocas veces en el juego (tantas como tipos de sala hay).

3.5.2.2. Ampliar

El tamaño de una sala se mide en plazas, que significa el número de monstruos que puede haber dentro durante cada franja horaria. Las salas recién construidas disponen (generalmente) de una sola plaza. En cuanto el jugador tenga suficiente basura la sala puede ser ampliada para crear otra plaza más, aunque cada ampliación es exponencialmente más cara que la anterior, tanto en basura necesaria como en esfuerzo para completar la obra. El tamaño es importante sobretodo para las salas públicas, ya que cuantos más clientes haya en su interior, mayor es el beneficio obtenido.

3.5.2.3. Actualizar

Cada sala ofrece una característica particular que mejora algún atributo de los monstruos que la utilizan como clientes. Si se actualiza una sala se mejoran sus prestaciones. Por ejemplo, si un gimnasio de nivel 1 hace subir X puntos de fuerza a los monstruos cliente que se encuentren en su interior durante una franja horaria, otro de nivel 2 subiría X más un cierto incremento en las mismas condiciones, y otro de nivel 3 aun más.

La calidad del servicio ofrecido por la sala no solo depende del nivel de la misma, sino también de lo buenos que sean los monstruos empleados que trabajen en su interior (si es una sala que requiera de empleados para su funcionamiento). Por ejemplo, un gimnasio de cierto nivel puede ser más efectivo que otro de su mismo nivel si el monitor que trabaja en él es mejor, lógico si lo comparamos con la realidad, donde no solo cuenta la calidad del material existente sino también la profesionalidad con la que sea manejado.

3.5.3. Publicación

Las salas de una guarida pueden ser públicas o privadas. Inicialmente las salas son privadas, de forma que solo pueden ser utilizadas por las criaturas de la misma guarida (todo se mueve en el ámbito del mismo jugador). Sin embargo en algunas existe la posibilidad de abrirlas al público, es decir, permitir que criaturas de otros jugadores entren y utilicen sus servicios cobrándoles un precio regulable (figura 3.14). El precio de entrada puede ser modificado cuando se desee y será lo que paguen las criaturas ajenas por utilizar la sala. También se podrá escribir un texto publicitario que leerán los jugadores cuando inicien una búsqueda de salas públicas.

El buscador de salas públicas aparece cuando se intenta asignar una tarea a un monstruo en su lista de tareas y se elige una sala externa a la guarida. Se trata de un formulario que permite elegir el tipo de sala (*gimnasio, disco pub, balneario spa, etc*), franja horaria (predefinida a la seleccionada en la lista de tareas) y un *checkbox* para ver o no las salas completas (predefinido a no, ya que no se puede entrar en las salas completas). También aparecen opciones de filtro (rango mínimo y máximo tolerado) como precio, nivel o distancia.

Al realizar la búsqueda se muestra una lista de salas donde se mostrarán los siguientes campos: precio de entrada, beneficio estimado obtenido por franja horaria (resultado calculado a través del nivel de la sala, habilidad del personal empleado y distancia), distancia a la que se encuentra del monstruo en ese momento (cuanto más lejos esté la sala más se tarda en llegar y menos tiempo se

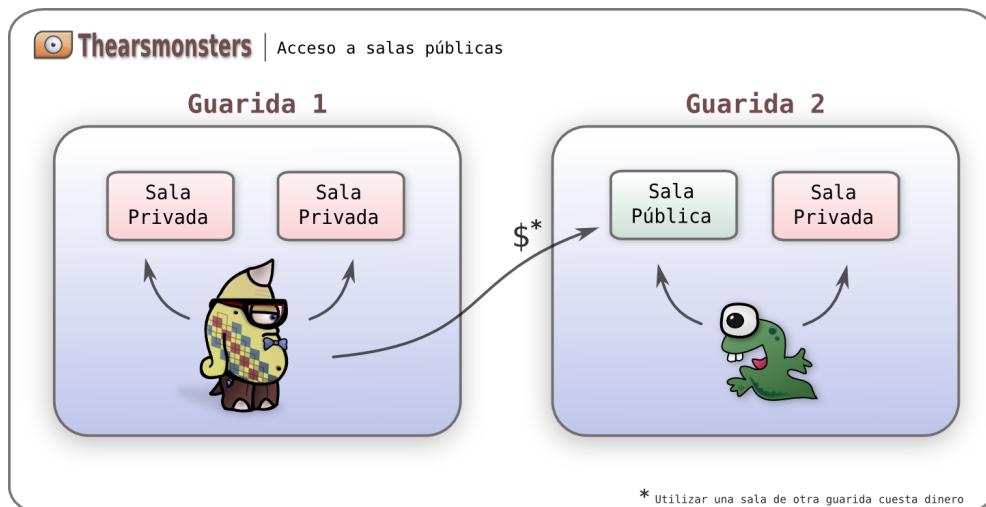


Figura 3.14: Los monstruos de una guarida pueden utilizar todas las salas de la misma. En otras guaridas solamente podrán acceder a las salas públicas, pagándole al dueño el precio establecido por el servicio. Así aparece un nuevo mercado dentro del juego.

aprovecha dentro de ella), mensaje publicitario del dueño, aumento de felicidad estimado, tamaño, nivel, suciedad y número de plazas libres de cada sala. Inicialmente se muestra el resultado ordenado por el beneficio estimado, aunque el nombre de cada campo es un enlace para ordenar la lista por ese campo (por ejemplo si se pulsa donde pone “precio de entrada” se ordenará la lista por el precio de entrada).

El carisma de los monstruos que atienden en las salas públicas es importante ya que los clientes aumentan su felicidad cuando visitan salas públicas de otras guaridas y son bien atendidos (algo similar a lo que sucede cuando las personas van a cenar a un restaurante).

3.5.4. Restricciones de acceso

Cuando se define una tarea nueva para un monstruo, hay que validar si el monstruo en cuestión puede estar en la sala donde pretende realizar esa tarea

(el modelo se encargará de validarla y la vista deberá ir ofreciendo solamente las posibilidades correctas). Para ello cada tipo de sala definen las edades que pueden entrar para cada tipo de tarea, los tipos de tarea que se pueden realizar en su interior (cliente, empleado, obrero, dormir, comer ...), la capacidad máxima de cada tarea, las restricciones por el número de plazas y si el acceso es público o no. Si alguna de estas condiciones no se cumple entonces no se podrá añadir la tarea a la planificación del monstruo.

3.5.5. Tipos de sala disponibles

Estos son los tipos de sala que incluye el juego, y determina el máximo número de salas que puede haber en una guarida, ya que solamente podrá haber una sala de cada tipo. Se diferencian unas de otras por su utilidad, o el tipo de tareas que se pueden realizar en ellas. La mayoría de las diferencias se establecen mediante atributos (como el número de clientes que puede albergar, sala publicable, máximo tamaño que puede alcanzar, etc). Los tipos de sala que puede haber en una primera aproximación son los siguientes¹:

- Básico:
 - **Ojo de la Vida:** Incuba los huevos de monstruos y alimenta a toda la guarida.
 - **Almacén de basura:** Permite extraer y almacenar basura.
 - **Oficina de Comercio:** Intercambio de basura por dinero y viceversa.
 - **Apartamentos Dormitorios:** Lugar donde duermen los monstruos. Define la cantidad de espacio vital y por consecuente el número de monstruos que puede haber en la guarida.

¹Realmente, en el análisis sería necesaria una explicación de los detalles de cada sala para que posteriormente puedan ser implementadas, sin embargo toda esta información no es necesaria en la memoria. Si se desea conocer el funcionamiento detallado de cada sala, basta con leer el manual en línea del juego o simplemente jugar a él, ya que incluye información integrada en la propia interfaz.

- Felicidad y Carisma:

- **Cultivo de Trufas:** Para cultivar deliciosas trufas que hacen felices a los monstruos.
- **Cocina:** Mejora la felicidad a la hora de comer y reduce las posibilidades de enfermar.
- **Cementerio:** Clasifica los muertos en el *ranking* de leyendas y reduce la pérdida de felicidad tras las muertes de los familiares.
- **Comedor:** Mejora el carisma y permite alquilar el servicio de comida.
- **Portal de Entrada:** Mejora el nivel social de la guarida, y por consecuente el carisma.
- **Ocio:** Mejoran el carisma en función de número criaturas simultaneas y la felicidad si el local coincide con los gustos de la criatura cliente. Son la **Sala Cill Out**, el **Metal Pub**, **Disco Techno**, **Reggaeton Pub** y **Rock Indie Pub**.

- Atributos base:

- **Gimnasio:** Mejora la Fuerza Base.
- **Balneario SPA:** Mejora la Vitalidad Base.
- **Tatami de agilidad:** Mejora la Agilidad Base.
- **Templo de meditación:** Mejora Inteligencia Base.

- Salud:

- **Sanatorio:** Cura enfermedades de manera rápida y eficaz.

- Habilidades:

- **Aula de enseñanza:** Lugar donde se enseñan unos monstruos a otros las habilidades aprendidas.

- **Sala de entrenamiento individual:** Para mejorar habilidades de batalla sin necesidad de maestro.
- Cría:
 - **Guardería infantil:** Se mejoran rápidamente todos los atributos base de las crías.
 - **Parque infantil:** Mejora la felicidad de las crías de manera permanente.
 - **Sala del amor:** Para que los monstruos se reproduzcan y no haya que comprar huevos.

3.6. Monstruos

Los monstruos son el motor del juego: construyen salas, las amplían, las mejoran, las utilizan, realizan misiones, luchan en batallas, consiguen premios, recolectan basura, cultivan hongos, etc. Para realizar satisfactoriamente sus tareas encomendadas deberán estar bien alimentados, felices y mejorar constantemente sus habilidades, bien sea practicándolas o a través del aprendizaje en las aulas de enseñanza. La estructura básica de los monstruos se muestra en la figura 3.15.

Hay diferentes razas de monstruos con sus ventajas e inconvenientes asociados. Cada raza tiene unas características genéticas, que determinan el valor inicial de los atributos al nacer. Posteriormente se puede mejorar los atributos de la criatura y sus habilidades utilizando las salas oportunas para cada atributo.

3.6.1. Estado de un monstruo

El estado de un monstruo evoluciona constantemente. Depende de todos sus atributos, que son bastantes y que se actualizan después de cada turno. Los atributos se dividen en varios niveles, de forma que hay unos pocos básicos que sirven para construir otros compuestos.

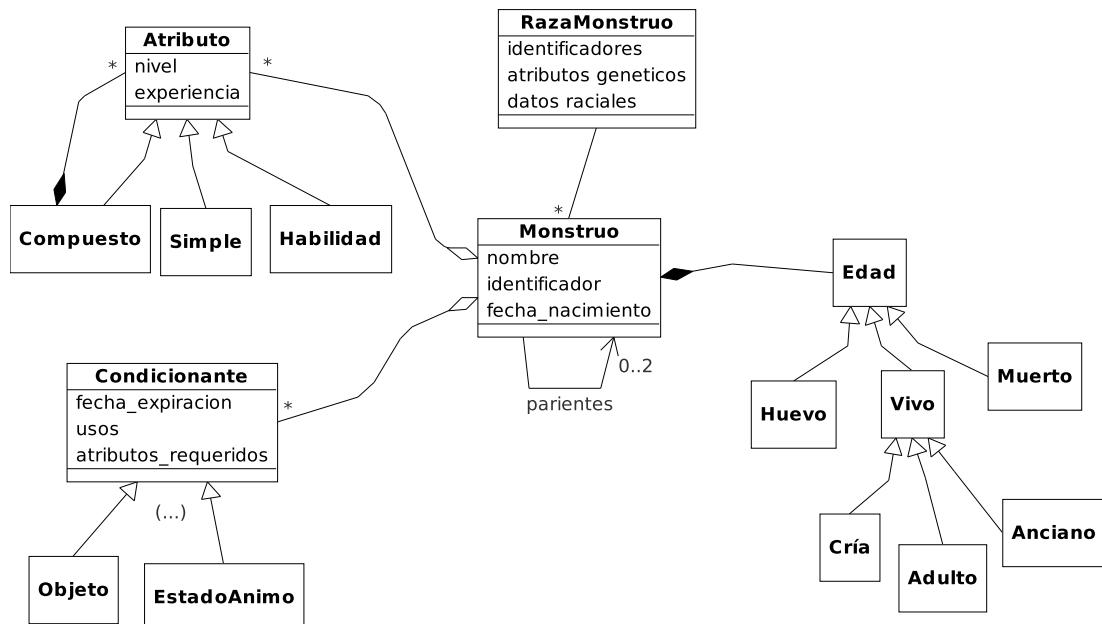


Figura 3.15: Diagrama de clases con la estructura de los monstruos del juego, sin entrar en detalles.

3.6.1.1. Características genéticas raciales

Diferencian una raza de otra y determinan el estado inicial de una criatura al nacer. Están ligadas a la raza, de modo que no cambian a lo largo del ciclo de vida.

- **Identificadores:** Identifican la raza y facilitan su comprensión.
 - **Nombre:** Nombre de la raza de monstruo
 - **Imagen:** Ilustración de un monstruo de esa raza
 - **Descripción:** Breve texto que describe los aspectos más importantes de la raza.
- **Atributos genéticos:** Definen el valor inicial de los atributos base cuando la criatura nace (nivel base, que luego se podrá mejorar).

- **Agilidad inicial**
 - **Fuerza inicial**
 - **Vitalidad inicial**
 - **Inteligencia inicial**
 - **Carisma inicial**
- **Datos raciales:** Datos que describen las necesidades y virtudes de cada raza.
- **Espacio vital:** Sitio libre necesario para que el monstruo viva en la guarida, si no hay espacio suficiente la criatura no saldrá del huevo. Este dato es muy importante porque el espacio total en una guarida determina el número máximo de monstruos que se puede tener. Representa la calidad de la raza, de forma que razas mejores exigen más espacio vital.
 - **Alimentación diaria:** Cuanto debe consumir diariamente para mantener la alimentación de la criatura a nivel medio (es decir, con valor=1). Para mejorar la alimentación a partir de ese punto la necesidad diaria de comida aumentará exponencialmente.
 - **Esperanza de vida:** Tiempo aproximado que vivirá el monstruo desde que nace hasta que muere, contando todas las etapas de su vida (cría, adulto y anciano).
 - **Tipo de raza:** Clasifica la raza en un tipo determinado, dependiendo de sus cualidades físicas. Algunos ataques o bonificadores afectan de distinta forma a cada tipo de raza. Existen los siguientes tipos de monstruo
 - *Mollusca*: Invertebrados, blandos, con tentáculos, amebas, etc. Tienen ataque adicional a los monstruos voladores.
 - *Crustacea*: Con garras, patas en pico, caparazones, etc. Tienen ataque adicional contra Vermes.

- *Verme*: Gusanos, reptadores, cilíndricos. Tienen ataque adicional contra Humanoides. Se recuperan más rápido de las enfermedades.
- *Humanoide*: Con brazos y manos, patas, manos, etc. Tienen ataque adicional contra Crustaceas. Son mejores dando clases.

Además pueden ser voladores o terrestres. Los voladores ganan agilidad cuanto más grandes sean las salas donde luchan, son más rápidos en los desplazamientos pero no pueden mejorar su fuerza en el gimnasio.

- **Fertilidad**: Capacidad para reproducirse. Cuanto mejor sea la fertilidad menos intentos serán necesarios para producir un huevo.
- **Sistema inmunológico**: Resistencia a las enfermedades y a alimentos en mal estado. Normalmente las razas de mayor tamaño son más débiles en este aspecto.
- **Suciedad**: Cuanto ensucia o estropea una sala al utilizarla. Monstruos más sucios dificultan más las tareas de mantenimiento en las salas de la guarida. Convivir con monstruos sucios eleva las probabilidades de contraer enfermedades.
- **Tiempo de metamorfosis**: Tiempo necesario dentro del capullo para realizar la metamorfosis de cría a adulto.

3.6.1.2. Atributos simples

Son los atributos que se pueden mejorar directamente, las cualidades de la criatura se componen a partir de ellos. Para mejorar un atributo simple hay que utilizar la sala correspondiente.

- **Fuerza base**: Permite llevar más carga al transportar cosas. Hace más daño en las batallas al impactar. Se mejora en la Sala de Pesas.
- **Agilidad base**: Desplazamientos por el escenario más rápidos. Mayor probabilidad de atacar primero en las batallas. Se mejora en el Tatami de Agilidad.

- **Vitalidad base:** Mejor aguante ante las adversidades (enfermedades, venenos, impactos). Se mejora en el balneario-spa y tras padecer alguna enfermedad.
- **Inteligencia:** Mayor rapidez aprendiendo y mejorando habilidades. Más probabilidades de hacer un crítico en la batalla. Se mejora en el Templo Meditación.
- **Carisma:** El carisma es la “buena presencia”, “habilidad social” o “saber estar”. Mejora los atributos de los aliados en la batalla si es la criatura líder. Cuando un monstruo utiliza una sala de otra guarida aumentará su felicidad tanto como carisma tenga el monstruo anfitrión. Por lo tanto si los monstruos que trabajan en público tienen buen carisma, más criaturas ajena querrán visitar las instalaciones y pagarán por ello, así que el carisma repercute directamente en la economía de una guarida. Se mejora en algunas salas donde se reunen varios monstruos.
- **Felicidad:** Depende principalmente de la cantidad de trufas que se hayan cultivado la última semana de juego. También mejora visitando los locales favoritos del monstruo (Disco, Chill-Out, Rock, etc) o utilizando salas de otras guaridas (dependiendo del carisma del monstruo que atienda en la otra sala), aunque este tipo de mejoras son temporales (al cabo de un tiempo se recupera el nivel de felicidad anterior). La felicidad influye en multitud de aspectos de la vida diaria de un monstruo, llegando incluso a considerarse el atributo simple más importante.

3.6.1.3. Atributos compuestos

Definen el comportamiento del monstruo en la batalla y la capacidad para realizar tareas. El valor de un atributo compuesto puede ser la combinación de atributos simples y/o de otros atributos compuestos. Normalmente el valor está condicionado por el factor de la habilidad correspondiente, de forma que antes de aprender la habilidad el valor del atributo es cero (por ser la habilidad

de nivel cero también). En la interfaz del juego solamente deben mostrarse los atributos específicos con valores diferentes de cero.

La lista de atributos compuestos es bastante grande (hay un atributo compuesto por cada tarea necesaria en la guarida). Ejemplos de atributos compuestos: **construcción, recolección, docencia, liderazgo**, etc.

3.6.1.4. Habilidades aprendidas

Son las habilidades que lleva aprendidas el monstruo en ese momento. Hay dos tipos de habilidades: las de *Trabajo* y las de *Batalla*. Las habilidades de trabajo son las que sirven para ofrecer un mejor servicio en alguna tarea de la guarida, como atender en una sala o construir. Este tipo de habilidades se aprenden realizando la propia tarea para la que sirven (se mejoran solas con la práctica) y las puede hacer cualquier criatura de cualquier raza. Las habilidades de batalla disponibles son diferentes para cada raza y para ser mejoradas necesitan ser practicadas en la sala de entrenamiento. Ambos tipos de habilidades también pueden ser mejoradas mucho más rápido si hay alguna criatura que las enseñe en un aula de enseñanza. Un monstruo que enseña una habilidad sólo puede hacerlo hasta el nivel en el que se encuentra dicha habilidad, es decir, que un alumno solo puede aprender la habilidad hasta el nivel de su maestro, si quiere seguir mejorándola tendrá que practicar en solitario o bien acudir a otro maestro.

Por cada nivel de cada habilidad hay una barra de experiencia que se va llenando a base de sesiones de aprendizaje (franjas horarias practicando o asistiendo a clases) hasta que se alcance el próximo nivel, momento en que se vuelve a vaciar la barra. La cantidad de experiencia que tiene la barra es una constante predefinida, y no aumenta con cada nivel. Asistir a clases es más costoso pero más rápido que aprender practicando.

3.6.1.5. Condicionantes

El estado de un monstruo puede verse afectado por los condicionantes (que pueden ser objetos como armas, o simplemente estados de ánimo).

Los objetos se ganan en los templos después de ganar ciertas misiones. Un objeto es un decorador de un monstruo, que le permite tener mejores atributos, más nivel en algunas habilidades o bien realizar cosas que antes no podía. Pueden ser armas, pócimas, amuletos, etc. Cada objeto tiene una serie de restricciones (atributos requeridos) que permiten poseerlos solamente a ciertos monstruos (requieren cierta cantidad de fuerza, agilidad, vitalidad, inteligencia, carisma, o están solo disponibles para ciertos tipos de raza). Además tienen una duración limitada, al cabo de la cual desaparecen. Pueden ser transferidos entre diferentes monstruos (que cumplan las condiciones de pertenencia) y también eliminados cuando se deseé.

Los estados de ánimo pueden ser por ejemplo la alegría de haber asistido a un evento o el haber ganado algún premio en una misión. Por ejemplo las salas tipo discoteca aumentan la felicidad de los monstruos durante un tiempo determinado, pues este comportamiento se puede modelar a modo de condicionantes.

También pueden ser condicionantes las enfermedades, las heridas causadas después de un combate y casi cualquier cosa que pueda alterar temporalmente el estado del monstruo.

3.6.2. Ciclo de vida de los monstruos

En casi cualquier juego tradicional de estrategia, las unidades de batalla se crean y duran hasta que alguien las elimine en combate. En el juego *Thearsmonsters* sucede justo al revés: ningún monstruo será eliminado en una batalla (tan solo se le aplicará algún condicionante negativo a modo de heridas causadas), y aunque no peleen nunca tienen una esperanza de vida limitada.

La esperanza de vida es un atributo racial muy importante, debido a que los monstruos van mejorando sus habilidades a medida que pasa el tiempo. Cuanto más tiempo vivan, más podrán mejorar una habilidad (claro que también depende del atributo *inteligencia*, porque cuanto más inteligentes, más rápido aprenden).

Como puede apreciarse en la figura 3.16, los diferentes estados por los que puede pasar un monstruo son: huevo, cría, adulto, anciano y muerto. El diagrama

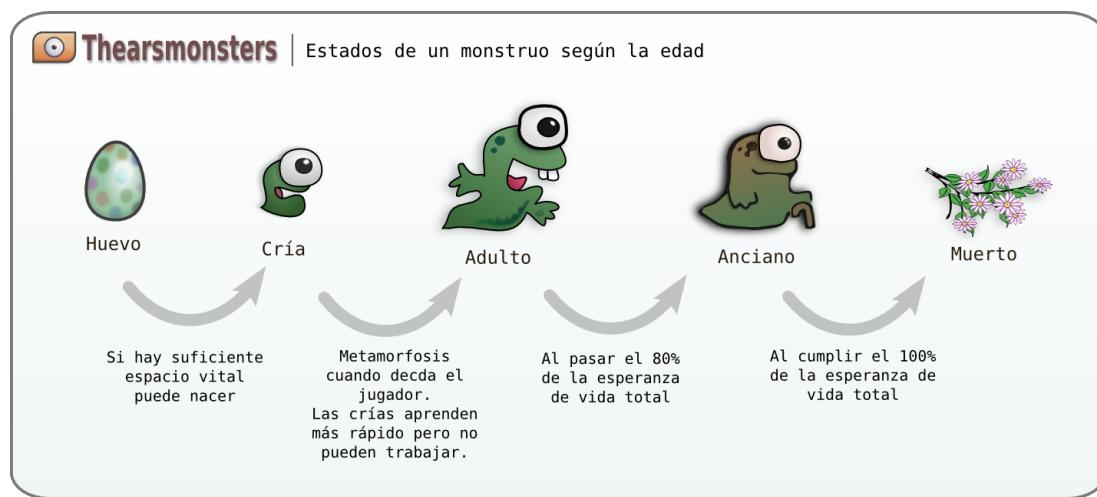


Figura 3.16: Diferentes estados por los que pasa la vida de un monstruo.

de estados es tan simple que hasta resulta intuitivo. Lo que quizás no sea tan intuitivo es la forma en la que los jugadores podrán actuar sobre él.

Los jugadores consiguen huevos de monstruo (bien comprándolos o bien mediante la *sala del amor*) y deciden el momento en el que nacen. Desde el nacimiento, el monstruo ya tiene contados sus días de vida (la *esperanza de vida* va ligada a cada raza de monstruo). El paso de cría a adulto es indeterminado, y se realiza mediante la *metamorfosis* cuando el jugador lo vea conveniente (las crías aprenden nuevas habilidades mucho más rápido que los adultos, sin embargo no pueden trabajar), sin embargo tanto si es cría como adulto, cuando pase el ochenta por ciento de la esperanza de vida, el monstruo se convertirá automáticamente en anciano. Mientras es anciano el monstruo solo puede trabajar en muy determinadas tareas (como cuidar crías en la guardería o dar clases en el aula de enseñanza). Después de eso se cumple el cien por cien de la esperanza de vida y el monstruo muere.

Por lo tanto es muy importante que el jugador sea capaz de transmitir los conocimientos entre generaciones de monstruo, y para ello se cuenta con las *aulas de enseñanza* (las salas disponibles se enumeran en la sección 3.5.5), siendo este uno de los aspectos más importantes del juego.

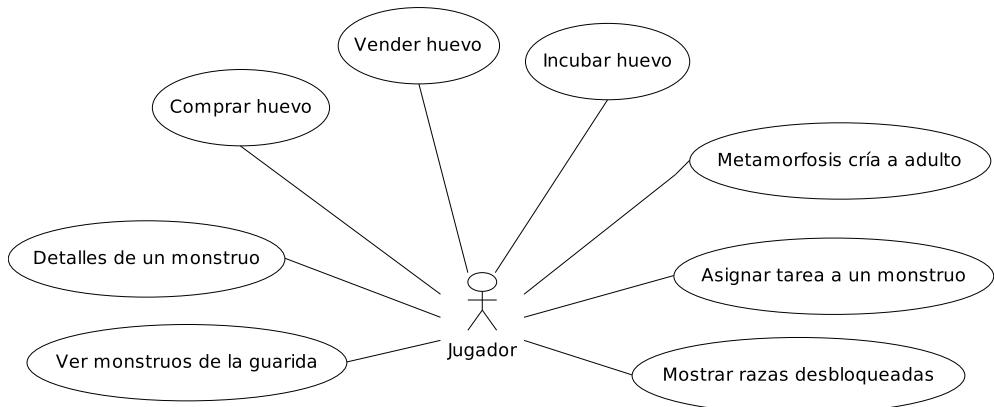


Figura 3.17: Diagrama de clases con la estructura de los monstruos del juego, sin entrar en detalles.

3.6.3. Monstruos en acción

Como ya se ha comentado en el apartado 3.2.1, los monstruos actúan según su lista de tareas diarias. Con respecto a los monstruos, los jugadores solo pueden ejecutar los casos de uso que se indican en la figura 3.17. Una vez que el jugador asigna las tareas, el sistema se encarga tras cada franja horaria de actualizar el estado del juego. Una tarea puede ser superpuesta por la realización de una misión, aunque solamente por un día, ya que la realización de las misiones se hace solamente una vez, en cambio las tareas se repiten diariamente.

En la lista de tareas de un monstruo debe haber obligatoriamente una serie de franjas horarias dedicadas a dormir (en los dormitorios) y a comer (en el ojo de la vida). La tarea por defecto es *vaguear en el ojo de la vida*, que se asigna automáticamente cuando no hay otra cosa que hacer. La lista de tareas inicial (cuando la criatura nace) tiene las horas obligatorias para descansar y para comer y el resto se rellena con la tarea por defecto.

Bajo este contexto, los jugadores se encargan de gestionar las tareas y el crecimiento de los monstruos que hay en su guarida, después el sistema se ocupa de que las tareas se ejecuten cuando sea preciso (sin necesidad de que el usuario esté conectado el juego).

3.7. Misiones

Los templos sirven para avanzar en la historia del juego, desbloquear nuevas características y conseguir premios. Se componen de una serie de salas que hay que superar sucesivamente hasta completarlas todas y de esa forma conseguir el premio final. En cada sala del templo se proponen una serie de objetivos que se deben cumplir para poder acceder a la siguiente sala. Al completar el templo entero se desbloquearán nuevos templos con objetivos aun más ambiciosos.

Además de la inevitable competencia que aparecerá al reflejar la calidad de la guarida de los jugadores en los *rankings*, existen lugares donde los jugadores pueden competir.

Los *Templos PvP* tienen normas específicas para cada competición. En común, siempre pueden inscribirse diferentes jugadores que cumplan unos requisitos (como puede ser un tamaño de guarida determinado, el haber completado cierta misión, el tener cierta raza en posesión, etc) y los ganadores se llevan un premio, que puede ser un objeto o simplemente dinero. Los perdedores al menos mejoran la vitalidad de sus criaturas al recuperarse de sus heridas.

También hay *Templos PvP* en el que compiten alianzas. El premio tiene que ser repartido entre los miembros de la alianza por el líder del grupo que compitió. Los criterios para formar un grupo de combate y repartir los premios son libres para cada alianza, lo único que se limita es el número de plazas que pueden ser llenadas para cada combate.

3.7.1. Realización de asaltos en una misión

Realizar una misión no es parte de la lista de tareas diarias de un monstruo, el motivo es porque los asaltos a las misiones se realizan solamente una vez (no es algo que se repita diariamente), y además porque es más sencillo de controlar desde la interfaz si se puede crear un grupo de monstruos y enviarlos a la batalla que tener que ir uno por uno modificando su lista de tareas. Aunque las batallas se computan de forma casi instantánea en el servidor, a nivel conceptual llevan

cierto tiempo, sobre todo por el desplazamiento hipotético que deben realizar los monstruos hasta llegar a los templos de las misiones. Así para realizar una misión lo que hay que hacer es seleccionar el conjunto de monstruos que van a participar en ella y elegir una franja horaria, tal y como se indica en el diagrama de actividades de la figura 3.18. Durante esta franja horaria no podrán realizar la tarea que tenían programada, pero su plaza seguirá reservada en el lugar de destino, y si esa plaza era en una sala externa habrá que pagar lo mismo (porque el jugador que ofrece el servicio está perdiendo una plaza). Tiene que ser así porque las tareas se repiten diariamente, y aunque en el momento de realizar la misión se pueda perder la tarea, al día siguiente sí que se realizará. Si algún monstruo no puede hacer la misión indicada en ese momento, el sistema deberá notificar al usuario antes de que termine de crear el grupo de asalto. Por ejemplo, el jugador elige tres monstruos de entre los que hay en su guarida y quiere enviarlos a realizar una misión a las 8 de la tarde. El sistema descubre a través de la planificación de tareas diarias que uno de esos monstruos estará durmiendo y que otro estará trabajando en una sala que no puede funcionar sin su presencia, entonces el usuario es avisado de que no se puede programar esa misión para esa hora y los motivos, además de dar algún consejo al respecto si es posible.

Una vez planificada la misión que se va a realizar, en el momento para que se haya previsto se realiza automáticamente la batalla y se crea un informe con los resultados de la misma que el jugador podrá visualizar cuando se vuelva a conectar al juego. Los monstruos que luchan serán más efectivos si cuentan con unas buenas habilidades de batalla. Los condicionantes como la felicidad y la alimentación también influyen en gran medida a los atributos de combate, por lo tanto será bueno preparar el grupo de batalla antes de enviarlo al combate. Más adelante se hablará del sistema de batallas.

3.7.2. Templos y salas con PNJs

En los templos se realizan las misiones de los jugadores. En cada sala de los templos hay una serie de enemigos que eliminar, que tienen un estado constante

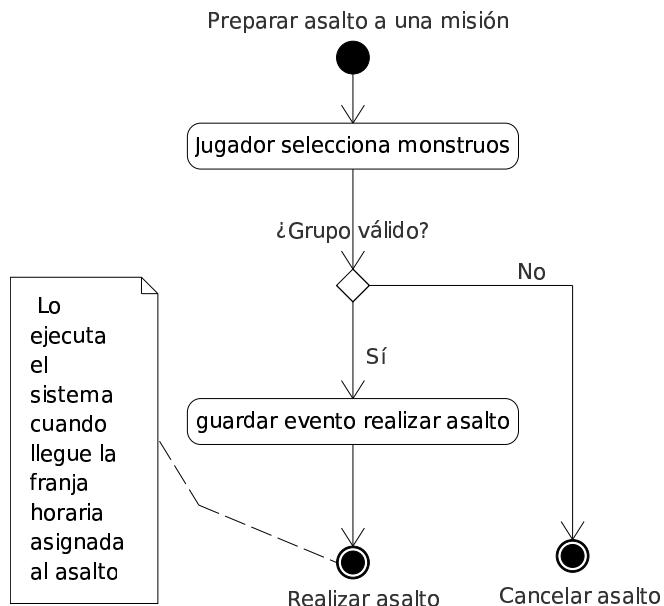


Figura 3.18: Proceso mediante el cual un jugador selecciona un grupo de monstruos para realizar un nuevo asalto a un tiempo.

(igual que el de los monstruos, pero con los atributos precisos para luchar y sus habilidades de batalla correspondientes). Si después de un asalto no se ha conseguido ganar la batalla, los PNJ volverán a renacer (este proceso es comúnmente llamado *respawning* en el mundo de los RPGs). Si se cumple la misión o se gana la batalla entonces la próxima vez que se entre en el templo será en la próxima sala, o bien el templo se completa por ser la última sala del mismo.

3.7.3. Sistema para desbloquear razas y salas

Cuando el jugador se registra y entra por primera vez en el juego, dispone de su guarida con solamente dos salas: el *ojo de la vida* y los *apartamentos*, que son lo mínimo para poder seguir adelante. Con el *ojo de la vida* se pueden incubar nuevos monstruos (gratis pero pequeños y debiluchos) hasta completar el poco *espacio vital* que aportan los *apartamentos* de una plaza de tamaño. Lo suficiente para realizar las primeras misiones (muy fáciles) y desbloquear así el *almacén*

de basura para poder construirlo y empezar a recolectar basura. A continuación se podrán ampliar las plazas de los apartamentos, y con más *espacio vital* se pueden criar más monstruos, que ayudarán a completar las siguientes misiones (cada vez más difíciles de superar), y así desbloquear la siguiente sala (la *oficina de comercio*) para poder cambiar basura por dinero. Con el dinero se podrán comprar otra raza de monstruo mejor, que ayuda a realizar las siguientes misiones . . . , y así sucesivamente se va mejorando la guarida.

Las primeras misiones son necesarias para desbloquear las distintas salas y razas de monstruo. Llegará un momento en el que se hayan desbloqueado todos los elementos que componen el juego, pero ahí no termina, porque luego se podrá acceder a los *templos PvP*, donde hay premios que ayudan a posicionarse mejor en el *ranking* de jugadores. Ahí es donde comienza realmente la experiencia multijugador (lo anterior sirve de tutorial a la vez que también es divertido).

A medida que se van completando las primeras misiones se va sucediendo una historia (que es la misma para todos los jugadores) relacionada con la religión verdadera del *Pastafarismo* cuyo Dios es el *Monstruo Espagueti Volador*², y así poder premiar al jugador con un nuevo capítulo del argumento mientras va completando nuevos templos.

3.8. Batallas

En una batalla se enfrentan dos grupos de monstruos el uno contra el otro. Como puede verse en el diagrama de la figura 3.19, una batalla se divide en rondas y cada ronda en turnos. En cada ronda se hacen parejas de monstruos contrincantes, proceso que se llama *emparejamiento* (para cada monstruo del grupo de menor tamaño se asigna aleatoriamente un contrincante del otro grupo, los que queden libres pasan de ronda sin luchar). Cada pareja de monstruos contrincantes lucha los turnos que hagan falta hasta que uno de los dos resulte derrotado. En

²No concierne a esta memoria la historia sobre la que trata el juego. Simplemente nombrar que hay un argumento pensado cuyo propósito es mantener el suspense en el jugador, además de atraer clientes potenciales que ya conozcan la satírica religión del *Pastafarismo*.

cada turno uno de los dos monstruos ataca y el otro defiende, teniendo mayor probabilidad de atacar el que tenga más agilidad. Cuando se hayan decidido todos los ganadores se pasa de ronda y se vuelve a hacer el emparejamiento. Se repite el proceso el número de rondas que haga falta, hasta que uno de los dos grupos resulte eliminado, que será el grupo perdedor.

3.8.1. Influencia de las habilidades de batalla

Algunas habilidades (como el Grito de batalla) se aplican una sola vez al comienzo de una batalla, otras (como el Movimiento maestro) al comienzo de cada ronda, y otras se aplican en cada impacto (como Probabilidad de crítico o Daño por crítico). Por ello antes de cada punto de los que se describen a continuación hay que aplicar las habilidades de batalla correspondientes.

3.8.2. Inicialización de la batalla

Se aplican los condicionantes del terreno (como la bonificación por tamaño de sala a los monstruos voladores) y de las habilidades correspondientes. Se inicializan los puntos de vida y se comienza con la primera ronda. El número de rondas en una batalla es indefinido, depende de lo igualados que estén los dos bandos, aun así siempre termina, debido a que en cada ronda se eliminan tantas criaturas como parejas de enemigos e hayan formado (siempre tiene que perder uno de las dos).

3.8.3. Inicialización de las rondas

En cada ronda se hacen parejas de enemigos (emparejamiento), es decir, que los monstruos pelean siempre uno contra uno, y no interfieren en las demás luchas, amenos que se aplique alguna habilidad que lo haga explícitamente. De cada una de estas luchas individuales tiene que haber un ganador, que pasa a la siguiente ronda con los puntos de vida que le queden.

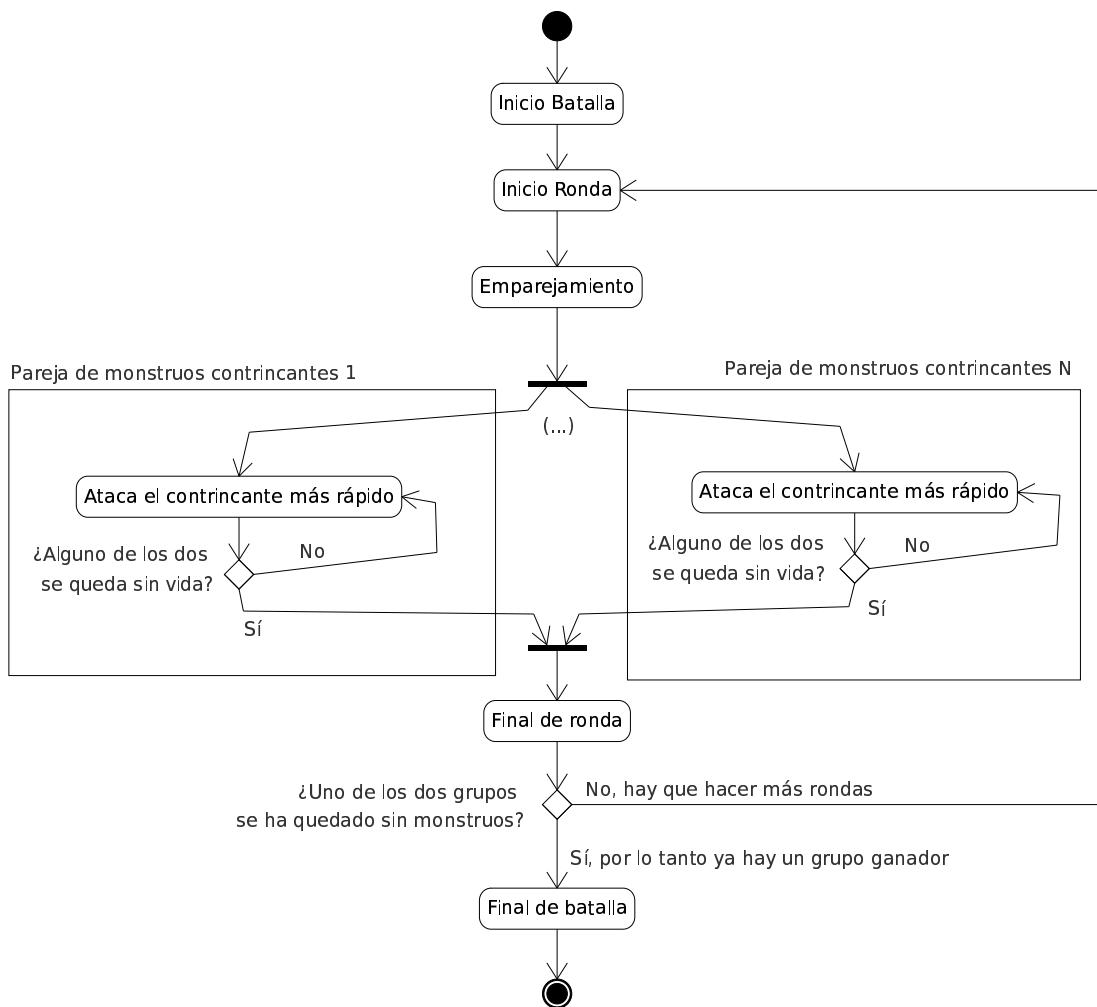


Figura 3.19: Realización de una batalla entre dos grupos de monstruos.

3.8.4. Factores de cada turno

Puntos de daño: Son el daño causado por impacto. En principio los puntos de daño son iguales a la fuerza del monstruo, aunque se pueden aumentar con habilidades ofensivas por parte del atacante y se pueden disminuir con habilidades defensivas por parte del defensor.

Puntos de vida: Al comienzo de la batalla los puntos de vida son 10 veces la vitalidad del monstruo, pero se van reduciendo en cada impacto recibido. Cada ronda termina cuando los puntos de vida de uno de los contrincantes se agota. El ganador pasa entonces a la siguiente ronda con los puntos de vida que le hayan quedado.

Puntos de agilidad: Se calculan en cada turno de la ronda como un número aleatorio entre cero y la agilidad de cada monstruo enfrentado. El que saque más puntos será el atacante, el otro será el defensor. El atacante impacta sobre el defensor y le resta tantos puntos de vida como puntos de daño sea capaz de provocar. En cada turno se vuelven a calcular aleatoriamente los puntos de agilidad para decidir quien ataca y se repite el proceso hasta que uno de los dos pierda todos sus puntos de vida. Es lógico pensar que cuanto mayor sea la agilidad de un monstruo mayor será la posibilidad de atacar en cada turno.

3.8.5. Finalización de la batalla

Si al terminar una ronda aún quedan monstruos en los dos bandos entonces se inicia la ronda siguiente. Si por el contrario uno de los grupos ha sido vencido entonces se termina la batalla. Las criaturas que han sido derrotadas no mueren, sino que aparecen con daños más o menos graves, a modo de enfermedad, que deberán ser tratados en la guarida de cada jugador, de forma que así ganarán más vitalidad para la siguiente ocasión (ya que la recuperación de una enfermedad mejora la vitalidad).

Capítulo 4

Diseño e Implementación

Índice general

4.1.	Subconjunto del análisis seleccionado para la primera versión	79
4.2.	Modelo y persistencia de datos	80
4.2.1.	Estructura de la base de datos	81
4.2.2.	Acceso a la base de datos	84
4.2.3.	Fachadas del modelo	87
4.3.	Conceptos modelados	90
4.3.1.	Usuarios	92
4.3.2.	Guardias	94
4.3.3.	Salas	96
4.3.4.	Monstruos	98
4.3.5.	Tareas	103
4.3.6.	Configuración	105
4.4.	Controlador y Vista	107
4.4.1.	Estructura condicionada a JSTL y <i>Apache Struts</i>	108
4.4.2.	SessionManager	110
4.4.3.	Acciones por defecto	113

4.4.4. Filtros Incluidos	115
4.4.5. <i>Custom Tags</i> para simplificar la vista	117
4.5. Internacionalización	122
4.5.1. Internacionalización de aplicaciones con <i>Struts</i>	123
4.5.2. Internacionalización y Localización con JSTL	123
4.6. Vistas interactivas con <i>JavaScript</i>	127
4.6.1. Importancia de la librería <i>JQuery</i>	128
4.6.2. Uso de <i>JavaScript</i> en <i>Thearsmonsters</i>	129
4.7. Acciones <i>Ajax</i>	131
4.7.1. Elección del método para implementar <i>Ajax</i> en <i>Struts</i>	131
4.7.2. Ejemplo de implementación con el método <i>Ajax</i> seleccionado . .	133

ESTE capítulo se centra en el diseño del producto software a partir del modelo conceptual obtenido del diseño del juego, que ya se ha abordado en los capítulos de Contextualización y de Análisis¹.

Se comienza seleccionando el subconjunto del concepto que es viable construir para la versión que corresponde a este proyecto. A continuación se abstraen las ideas y se dividen en capas que den soporte al desarrollo del software como pueden ser la persistencia, la lógica de negocio y la presentación. Siempre enfocado desde el punto de vista de la orientación a objetos, se van desarrollando los diferentes componentes del programa (usuarios, guardias, monstruos, etc)² y se van adaptando a las tecnologías utilizadas para después poder llevar a cabo la implementación.

¹Cuando hablamos de *diseño de videojuegos* [24] nos referimos al concepto, al “qué”. Desde el punto de vista del software, esto se corresponde realmente con la fase de análisis.

²Al igual que en el capítulo de análisis, los conceptos se explican mediante diagramas UML, solo que esta vez se tienen en cuenta las responsabilidades de cada clase y se disponen para facilitar la implementación.

4.1. Subconjunto del análisis seleccionado para la primera versión

Como hemos visto en el capítulo anterior, el juego *Thearsmonsters* está compuesto de conceptos como: cuentas de usuario (registro y autenticación), administración, alianzas, mensajes, guaridas, salas de la guarida, monstruos, tareas, misiones individuales, misiones multijugador, torneos, objetos (armas y amuletos), compra/venta de objetos, alquiler de servicios o comercio de parcelas para construir nuevas guaridas.

Gracias a la propia naturaleza del modelo conceptual, muchos de estos conceptos son prescindibles para conseguir una versión mínimamente jugable, es decir, que se pueda probar y permita la competencia entre jugadores. Para entender mejor lo que quiere decir “versión mínimamente jugable”, supongamos un juego de estrategia en tiempo real estilo *Age of Empires*, entonces esa versión es la que permita al usuario crear al menos el edificio principal y extraer al menos un tipo de recurso para poder entrenar al menos un tipo de unidad y poder atacar al contrario. De esta forma, aunque el juego sea muy simple, ya se pueden ir viendo sus características, y aumentarlas sería cuestión de añadir nuevos tipos de recursos, más unidades, más edificios, más mapas, etc. Es decir, que resulta bastante sencillo hacerse a la idea de como sería la versión final³.

Los conceptos que se van a incluir en la primera versión del juego son los que se muestran en la figura 4.1, y algunos conceptos no se implementan al completo. Por ejemplo, no se van a implementar todos los tipos de sala que se describen en el análisis, sino solamente aquellos que permitan el desarrollo dentro del contexto de esta versión (si no hay batallas, entonces no es necesario construir una sala de entrenamiento).

Gracias a esta decisión, se puede enfocar el esfuerzo dedicado al desarrollo a conseguir resultados más profundos, con mejor calidad, más mantenibles y con

³Aunque tratándose de un MMORPG realmente no podemos hablar de versión final ya que en este tipo de juegos el desarrollo es continuo e indefinido.

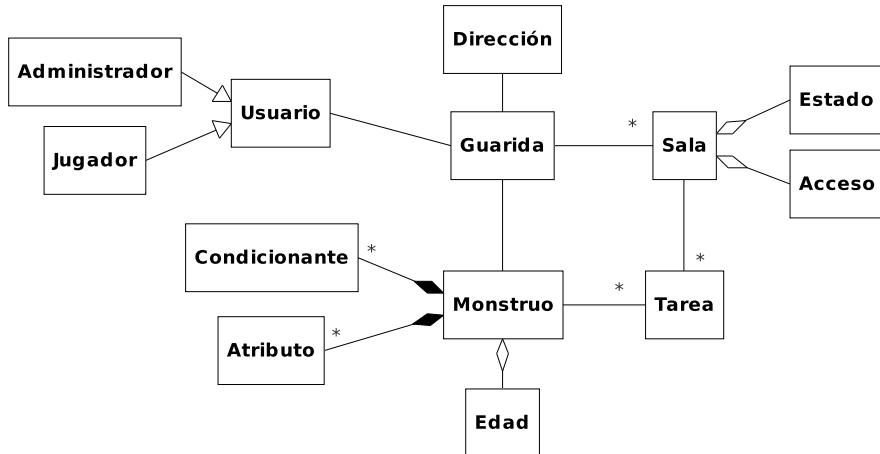


Figura 4.1: Conceptos para implementar en la primera versión del juego.

más posibilidades de futuro. Si por el contrario se hubiese optado por incorporar un mayor número de funcionalidades, sin preocuparse mucho por la flexibilidad del software o de la presentación, seguramente el resultado sería un juego más completo pero menos sostenible, lleno de errores, aparentemente pobre en aspecto y dando la falsa sensación de estar terminado.

4.2. Modelo y persistencia de datos

Thearsmonsters es un juego persistente, es decir, que cuando el usuario se desconecta, el mundo virtual del juego continúa funcionando. La persistencia de datos se realiza con una base de datos, y siguiendo el patrón arquitectónico *MVC* (Modelo, Vista, Controlador [32]) lo mejor será encapsular el modelo y su persistencia bajo una serie de sencillas interfaces a las que llamaremos *fachadas* [23].

En este capítulo se irán explicando los diferentes conceptos que hay que implementar en base al mismo esquema dividido en capas.

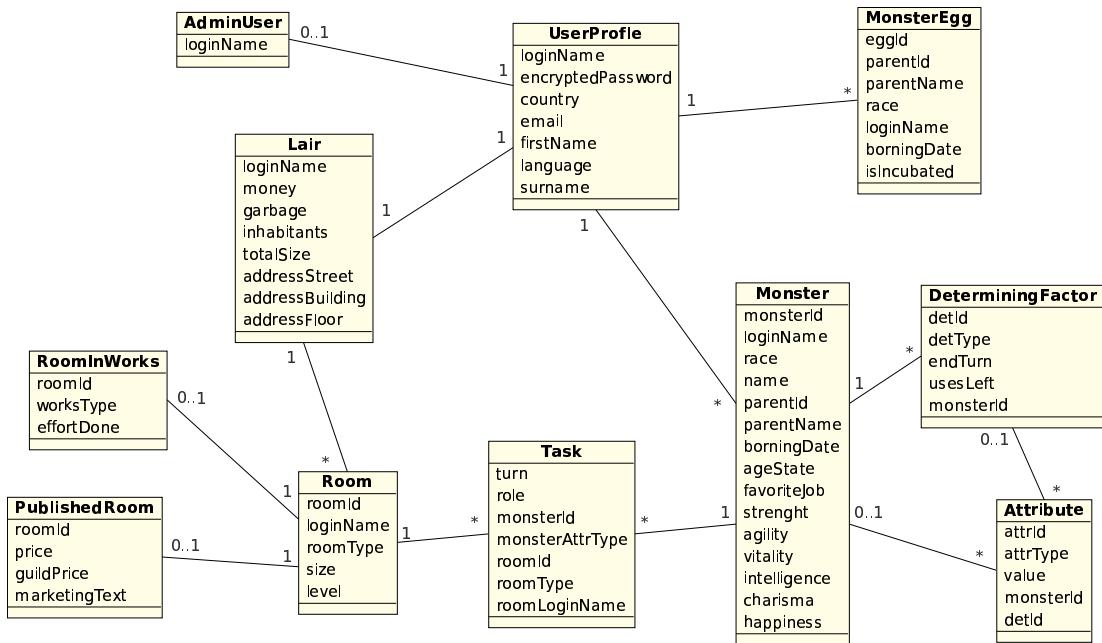


Figura 4.2: Estructura de la base datos relacional que da soporte a la persistencia de datos en *Thearsmonsters*.

4.2.1. Estructura de la base de datos

El sustento de la persistencia de datos se encuentra en la base de datos. El diseño de esta capa depende también de cómo se quieran modelar cada uno de los conceptos (por suerte no depende demasiado del tipo de base de datos concreto). En la sección siguiente (4.2.2) se verá cómo se accede a la base de datos, y más adelante, en la sección 4.3, se muestra cómo se realiza la traducción entre la capa de la base de datos y los objetos representados en memoria. Teniendo en cuenta esto, lo que se muestra aquí no es más que un resumen de la estructura global de la base de datos.

En la figura 4.2 se muestra una representación en UML de las tablas presentes en la base de datos. Cada tabla tiene el siguiente significado:

- **UserProfile:** Cada uno de los usuarios del sistema, tanto jugadores como administradores. La clave primaria es `loginName` (en todas las tablas de

la figura menos en `Task`, el atributo superior es la clave primaria), el resto son datos del usuario. Esta tabla es la raíz de todos los datos de usuario, es decir, que si se añaden restricciones de clave foránea para eliminación en cascada en el resto de tablas, entonces al eliminar la tabla `UserProfile`, se eliminarán también la guarida, los monstruos, las tareas y todo lo que esté relacionado con el usuario.

- **AdminUser:** Es un filtro para determinar qué usuarios tienen los privilegios de administrador. Después los administradores se identifican con su nombre de usuario y contraseña, igual que los jugadores.
- **Lair:** Datos de la guarida. Realmente estos datos y los del usuario podrían ir en la misma tabla (por la relación 1 a 1), sin embargo, debido al modelo conceptual y al significado de los mismos, es mejor mantenerlos separados. Además en la aplicación se recuperan en momentos distintos.
- **Room:** Salas de la guarida. En principio, como en cada guarida solo puede haber una sala de cada tipo (`roomType`), las salas quedarían identificadas con `loginName` y `roomType`, sin embargo, por motivos de flexibilidad, sencillez y eficiencia⁴ se identifican solamente con el atributo autoincrementado `roomId`. No es necesario guardar información dinámica distinta para cada tipo de sala, por lo tanto llega con identificar el tipo de sala (`roomType`) y así el modelo ya sabe qué clase debe instanciar (ya que el comportamiento de cada tipo de sala sí que es diferente).
- **RoomInWorks:** Hace de filtro para saber qué salas (`roomId`) están en obras, y guarda los datos necesarios para las mismas (tipo de obras y trabajo realizado).
- **PublishedRoom:** Hace de filtro para saber qué salas (`roomId`) están publicadas, y guarda los datos necesarios para las mismas (precio de entrada,

⁴La tabla `Room` tiene referencias desde `RoomInWorks`, `PublishedRoom` y `Task`, por lo tanto resulta más sencillo y eficiente realizar *joins* sobre de un solo atributo.

precio para los miembros de la alianza y texto publicitario).

- **Monster:** Datos de los monstruos de un jugador (`loginName`). Su clave primaria es `monsterId`. En esta tabla se guardan el nombre, la fecha de nacimiento (para conocer la edad actual del monstruo), el identificador del progenitor primario, el estado de la edad (cría, adulto, anciano), el trabajo favorito y el valor de los cinco atributos simples más importantes. Estos cinco atributos (fuerza, vitalidad, inteligencia, carisma y felicidad) salen en los listados de los monstruos, y resulta mucho más eficiente obtenerlos en la propia tabla que en la tabla **Attribute** (evita tener que hacer *joins*, aunque añade complejidad al código). El resto de atributos no se deben poner en esta tabla también porque se van ganando poco a poco (habría muchos atributos nulos), y además se irán añadiendo nuevos atributos a medida que salgan nuevas versiones del juego.
- **Attribute:** Almacena los atributos de los monstruos y de los condicionantes. La clave primaria es `attrId` para simplificar los *joins*, al igual que sucede con la tabla **Room**.
- **DeterminingFactor:** Son los condicionantes de un monstruo (decoradores como sentimientos, armas, amuletos, etc.) que pueden mejorar o empeorar el valor de sus atributos y alterar su comportamiento. También tiene atributos, que en este caso indican el tipo y el valor del atributo del monstruo que deben modificar.
- **MonsterEgg:** Los huevos de monstruo se guardan en esta tabla a parte, ya que en la implementación se representan con objetos distintos (aunque en el modelo conceptual un huevo no es más que un de un monstruo).
- **Task:** Las tareas que realizan los monstruos en las salas. En este caso, la clave primaria está formada por `monsterId` y `turn`, así ya se asegura la restricción de que un monstruo solamente puede realizar una tarea al mismo tiempo. Además esto no afecta al rendimiento, porque solamente se van a

hacer *joins* el atributo `monsterId` o `roomId`. El `role` indica el tipo de tarea (cliente, empleado, obrero, etc.), y el resto de atributos son redundantes (para mejorar el rendimiento de las consultas).

4.2.2. Acceso a la base de datos

Existen muchos métodos para implementar la persistencia de datos. La tendencia actual consiste en utilizar un *ORM*⁵, que se encarga de convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y el utilizado en una base de datos relacional. Para el lenguaje de programación *Java* existen varios, aunque el más utilizado es *Hibernate*. Sin embargo, hay programadores que siguen prefiriendo crear sus propias herramientas ORM porque utilizar cualquiera de estos *frameworks* también conlleva sus propios problemas. Para el desarrollo de este proyecto se ha optado por no utilizar ninguna herramienta ORM y manejar la persistencia utilizando *DAOs*⁶ [14], debido a que *Struts* no proporciona un ORM nativo⁷ y, a priori, será mucho más sencillo implementar la capa de persistencia “a mano” que tener que integrar y aprender a utilizar un ORM por cuenta propia.

Como se ve en la figura 4.3, un DAO se puede representar como una interfaz (con sus respectivos métodos para gestionar objetos persistentes). Dicha interfaz se implementa distribuyendo las responsabilidades entre una clase abstracta (*AbstractSQL.DAO*) y una serie de subclases. Los nombres que se han puesto en la figura 4.3 son genéricos, para cada DAO que se incluya en la aplicación se debe sustituir el símbolo “_” por el nombre del objeto persistente (por ejemplo, el SQL.DAO que accede a los usuarios se corresponde con SQLUserProfileDAO, cuya SQL.DAOFactory es SQLUserProfileDAOFactory).

⁵De las siglas inglesas *Object-Relational Mapping*, que significan “Mapeo Objeto-Relacional”.

⁶Los DAOs (*Data Access Objects*) son las clases encargadas de manejar la persistencia de los objetos. Lo malo es que suele ser necesario hacer un DAO distinto para cada tipo de objeto persistente.

⁷La mayoría de *frameworks* modernos integran por un ORM por defecto, por ejemplo *Ruby on Rails* utiliza *ActiveRecord* [7].

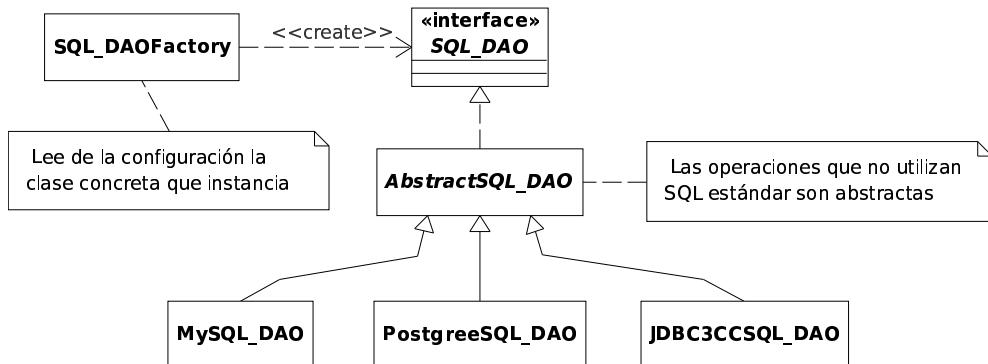


Figura 4.3: Estructura general que siguen los DAOs implementados en la aplicación.

Los objetos que se obtienen en los DAOs son sencillos contenedores de información, con métodos que simplifican sus cambios de estado. Estos objetos (persistentes gracias a los DAOs) siguen el patrón *Value Object*, y son *serializables* para poder ser compartidos en caso de que el despliegue de la aplicación se hiciese en un clúster de servidores.

En el diagrama de la figura 4.3, la factoría [19] se encarga de instanciar el DAO con la clase correspondiente (que lee de la configuración). En principio solamente se va a implementar una subclase (por ejemplo, si se utiliza una base de datos con el *driver JDBC3*, solamente es necesario implementar la subclase *JDBC3CCS_SQL.DAO*). Pero gracias a esta estructura, si en un futuro se desea cambiar la base de datos por otra (supongamos *PostgreSQL*) solo hay que crear la clase *PostgreSQL.DAO*, cambiar en la configuración el parámetro que le dice a la factoría cual es la nueva clase que debe instanciar y cambiar el *driver JDBC* que se conecta a la nueva base de datos.

Además, como la configuración se puede cambiar sin que haya que reiniciar la aplicación, será posible cambiar la base de datos por otra distinta sin detener la ejecución de la aplicación. Esto es un caso concreto del patrón *Business Delegate* [14], que también se utiliza para instanciar las distintas fachadas del modelo (cada fachada también tiene una factoría que lee de la configuración la clase concreta que debe instanciar), dándole al modelo entero la posibilidad de cambiar

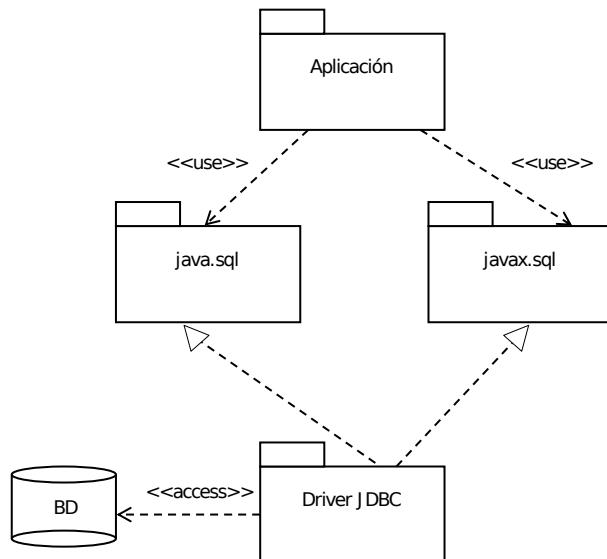


Figura 4.4: Driver JDBC. El programador siempre trabaja contra los paquetes *java.sql* y *javax.sql*, que forman parte de *Java SE*.

su implementación de modo transparente para los clientes.

Para implementar los métodos que acceden a la base de datos se utiliza JDBC (*Java DataBase Connectivity*), que es un API que permite lanzar *queries* a una base de datos relacional. Como se muestra en la figura 4.4, el programador trabaja contra los paquetes *java.sql* y *javax.sql*, que contienen un buen número de interfaces y algunas clases concretas, que conforman el API de JDBC. Para poder conectarse a la base de datos y lanzar *queries*, es preciso tener un *driver* adecuado para ella, que suele ser un fichero *.jar* donde se implementan todos los interfaces del API de JDBC.

Idealmente, si la aplicación cambia de base de datos, no es necesario cambiar el código; simplemente se necesita otro *driver*. Desafortunadamente, las bases de datos relacionales usan distintos dialectos de SQL (las principales diferencias son los tipos de datos y la generación de identificadores como secuencias o autonumerados), a pesar de que en teoría el SQL es un estándar. Por eso hay que dividir el código en fragmentos de SQL que son estándar y fragmentos que no lo son.

Para la estructura mostrada en la figura 4.3, los fragmentos estándar se escriben en la superclase (`AbstractSQL.DAO`), y los que no lo son se escriben de la forma que sea específica para cada base de datos en las subclases correspondientes. [23]

4.2.3. Fachadas del modelo

Dentro del contexto *MVC*, el modelo es el encargado de llevar a cabo la lógica de negocio y de manejar los datos del programa. El propio modelo debe también estar adecuadamente estructurado y tener bien definidas las responsabilidades de cada componente. [32]

Los servicios expuestos en el modelo deben ser similares a los casos de uso especificados en la fase de análisis, por eso se incluyen las *fachadas*⁸, que tienen únicamente los métodos necesarios para implementar cada *caso de uso* a modo de acciones, transaccionales o no, que crean, modifican, o destruyen objetos persistentes en términos de los DAOs. Por lo tanto el modelo se divide básicamente en dos capas independientes: las fachadas y los DAOs.

La figura 4.5 muestra la estructura de las fachadas del modelo. Los nombres de las clases que llevan el símbolo “_” describen la nomenclatura genérica que debe utilizar cada fachada concreta, por ejemplo, para la fachada de usuarios hacen falta las clases `UserFacadeDelegateFactory`, `UserFacadeDelegate`, `PlainUserFacadeDelegate` y una clase más por cada acción (método, caso de uso).

La factoría y la interfaz sirven, al igual que en los DAOs, para implementar un caso concreto del patrón *Business Delegate*, ayudando a desacoplar la capa de negocio de la presentación. La factoría lee de la configuración la clase concreta (`Plain_FacadeDelegate`) que debe instanciar, permitiendo un cambio de implementación sin tener que reiniciar el servidor. El cliente (en este caso el controlador) solamente necesita utilizar la factoría para instanciar un nuevo ob-

⁸En adelante, se llamará “fachada” al conjunto de clases que implementan el patrón *Session Facade* [14], que es una aplicación de otro patrón más general llamado *Facade* [19]. Una *Session Facade* representa un *workflow* y no persistencia, formado por un conjunto de casos de uso lógicamente relacionados.

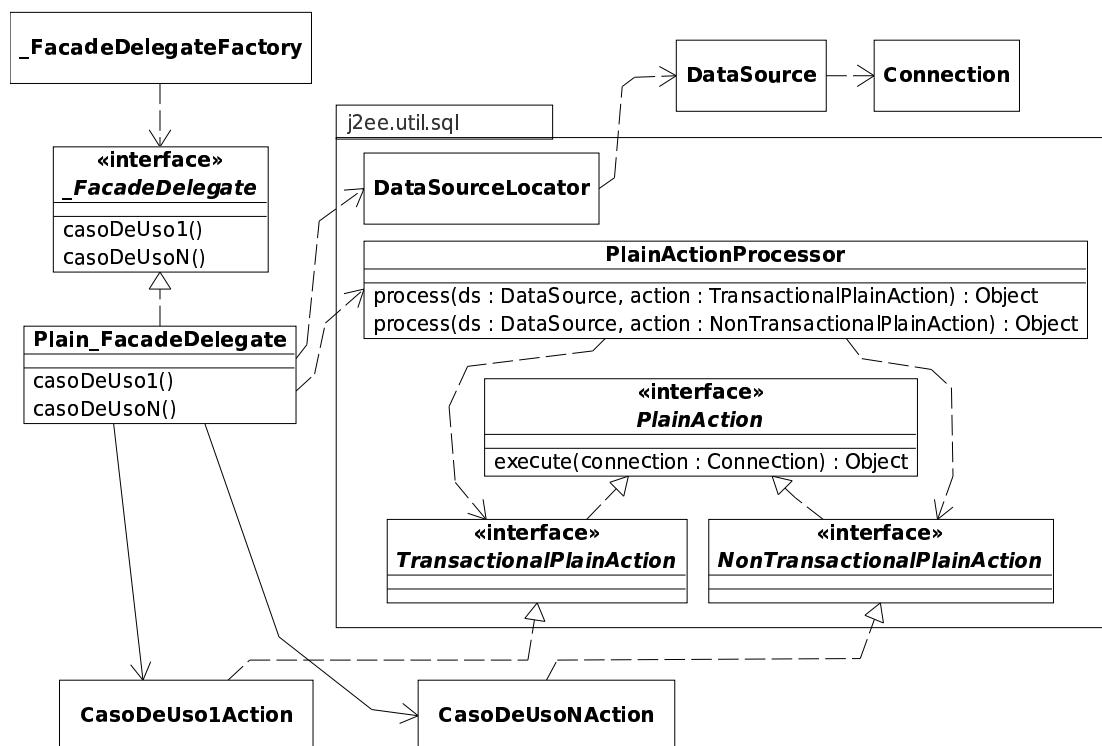


Figura 4.5: Estructura genérica de las fachadas del modelo junto con sus *PlainActions*. Las clases del paquete `j2ee.util.sql` son comunes a todas las fachadas.

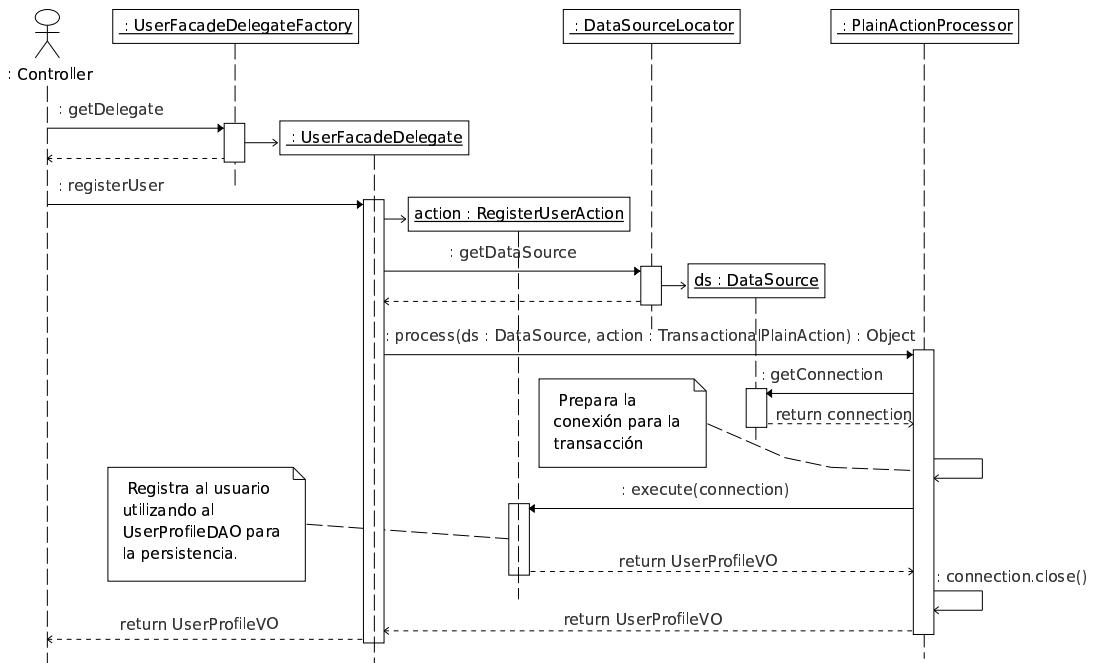


Figura 4.6: Ejemplo de ejecución de la acción `register_user` en la fachada de usuarios.

jeto que implemente la interfaz `_FacadeDelegate`), que es una fachada encargada de simplificar las acciones necesarias en unos pocos métodos más manejables y comprensibles.

La división de acciones en clases (una clase por acción) ayuda a manejar las transacciones de manera sencilla y ordenada. En el paquete `j2ee.util.sql` (del subsistema *standardutil*) se encuentran las clases necesarias para conseguir una conexión con la base de datos y para procesar acciones. El `DataSourceLocator` permite registrar y utilizar uno o varios `DataSources`, que sirven para obtener conexiones (generalmente implementa un *pool de conexiones* para no saturar la base de datos). Las acciones implementan el método `execute`, que recibe una conexión donde se van a realizar las consultas a la base de datos.

Como puede apreciarse en el diagrama de secuencia de la figura 4.6, la clase `PlainUserFacadeDelegate` utiliza el `DataSourceLocator` para obtener un `DataSource`

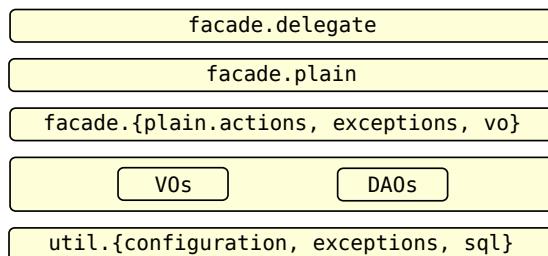


Figura 4.7: Implementación de la capa modelo aplicando a su vez el patrón arquitectónico *Layers*.

y pasárselo al `PlainActionProcessor` junto con la acción correspondiente a ese método (la acción puede ser de tipo transaccional o no). El `PlainActionProcessor` procesa de modo diferente las acciones transaccionales (que necesitan realizar varias consultas de modo transaccional) y las que no lo son. Lo que hace es obtener la conexión del `DataSource`, configurarla adecuadamente para realizar las transacciones y ejecutar la acción correspondiente.

Las acciones podrían realizar directamente las consultas a la base de datos, pero gracias a los DAOs (que se encargan de la persistencia de datos) solamente deben preocuparse de ejecutar la lógica de negocio. Esta división en capas, que puede apreciarse en el esquema de la figura 4.7, sigue las directrices del patrón arquitectónico *Layers*, logrando componentes independientes que favorecen el mantenimiento de la aplicación.

4.3. Conceptos modelados

En las secciones anteriores (4.2.2 y 4.2.3) se describe la estructura de las fachadas y de los DAOs del modelo de la aplicación. A continuación se van a describir cada uno de los conceptos del juego y cómo se modelan en base a estas dos capas anteriores.

En los próximos diagramas también se incluye la “capa de la base de datos”, aunque realmente no se trata una capa independiente del modelo ya que está bastante ligada a la implementación de los DAOs. Añadiendo este pequeño subcon-

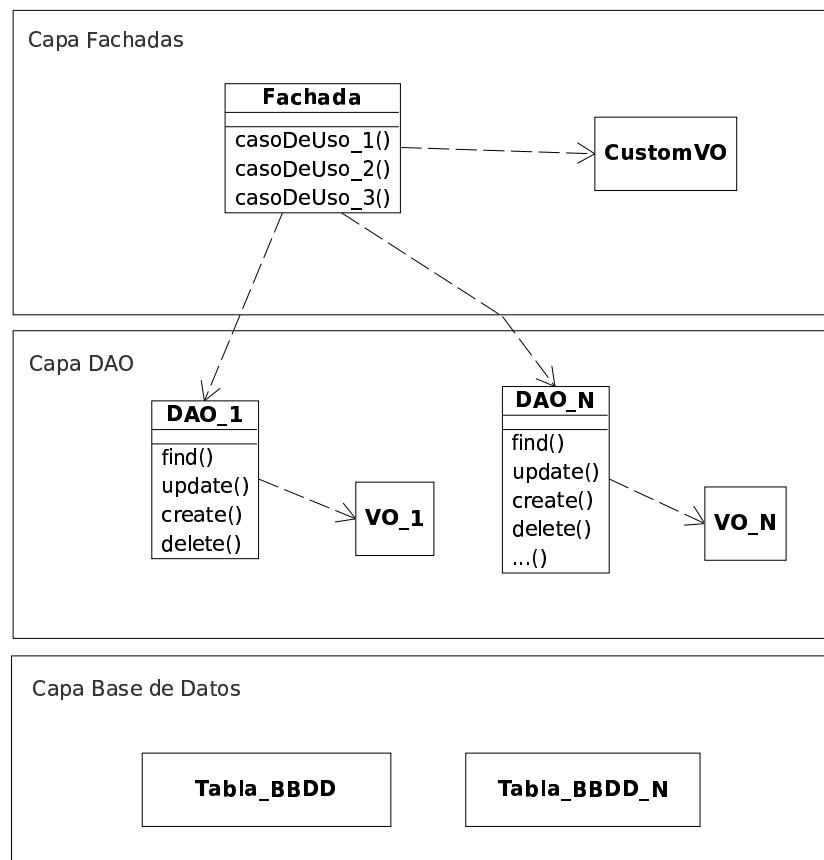


Figura 4.8: Estructura genérica en capas que sigue cada uno de los componentes del modelo.

junto del diagrama de la figura 4.2, resulta más fácil valorar la traducción de datos que deben realizar los DAOs entre la base de datos y el modelo.

En el modelo de la primera versión de la aplicación hay cuatro fachadas (para dividir los casos de uso en pequeños grupos) que son: usuarios, guarida, monstruos y tareas. También se crea otra fachada para las pruebas unitarias (casos de uso útiles en los test de *JUnit*).

En el diagrama 4.8 se puede apreciar el esquema general que se utiliza para modelar cada uno de los conceptos del juego. La implementación de cada fachada se encarga de realizar la lógica de negocio de cada caso de uso, utilizando a

los DAOs para la persistencia de datos (pueden utilizar uno o varios DAOs). Los métodos de las fachadas pueden devolver directamente el VO (*Value Object*) obtenido con el DAO correspondiente o bien encapsularlo en un *Custom VO* (para añadir más datos, para poder devolver varios VOs en el mismo método o para aplicar el patrón *Page by Page iterator*⁹ utilizando un *ChunkVO*).

A partir de ahora, en este tipo de diagramas se omiten los detalles de cada fachada o DAO (no es necesario mostrar las factorías, las interfaces, las acciones o el *PlainActionProcessor* entre otros). Además los métodos que aparecen en una fachada no tienen por qué ser todos los que hay, solamente se incluirán solamente aquellos relacionados con el resto del diagrama, para facilitar su comprensión.

A continuación se van explicando las decisiones de diseño que se han tomado para implementar cada uno de los conceptos importantes del sistema, todos ellos siguiendo el mismo esquema de la figura 4.8.

4.3.1. Usuarios

En la figura 4.9 se representa la parte del modelo encargada de registrar, identificar, modificar o eliminar cuentas de usuario. La representación no es exacta sino que se ponen solamente aquellos aspectos relevantes para su comprensión.

La clase **UserFacadeDelegate** es una representación de la fachada de usuarios (en realidad se compone de factoría, interfaz, implementación y una clase acción por cada método). Esta fachada tiene estado (que es una característica especial con respecto a las demás). Al principio solamente se pueden ejecutar los métodos **login** o **registerUser** que guardan el *loginName* en el estado de la fachada (en caso de éxito). Los demás métodos no necesitan en parámetro *login* porque lo adquieren de estado (por ejemplo, cuando se hace **change_password** se cambia la contraseña del usuario previamente identificado).

La clase **PasswordEncrypter** se utiliza para cifrar las contraseñas. De este

⁹El patrón *Page by Page iterator* encapsula un página (fracción) de una lista de elementos, con métodos adicionales para saber si hay más páginas y así poder recorrer toda la lista paso por paso.

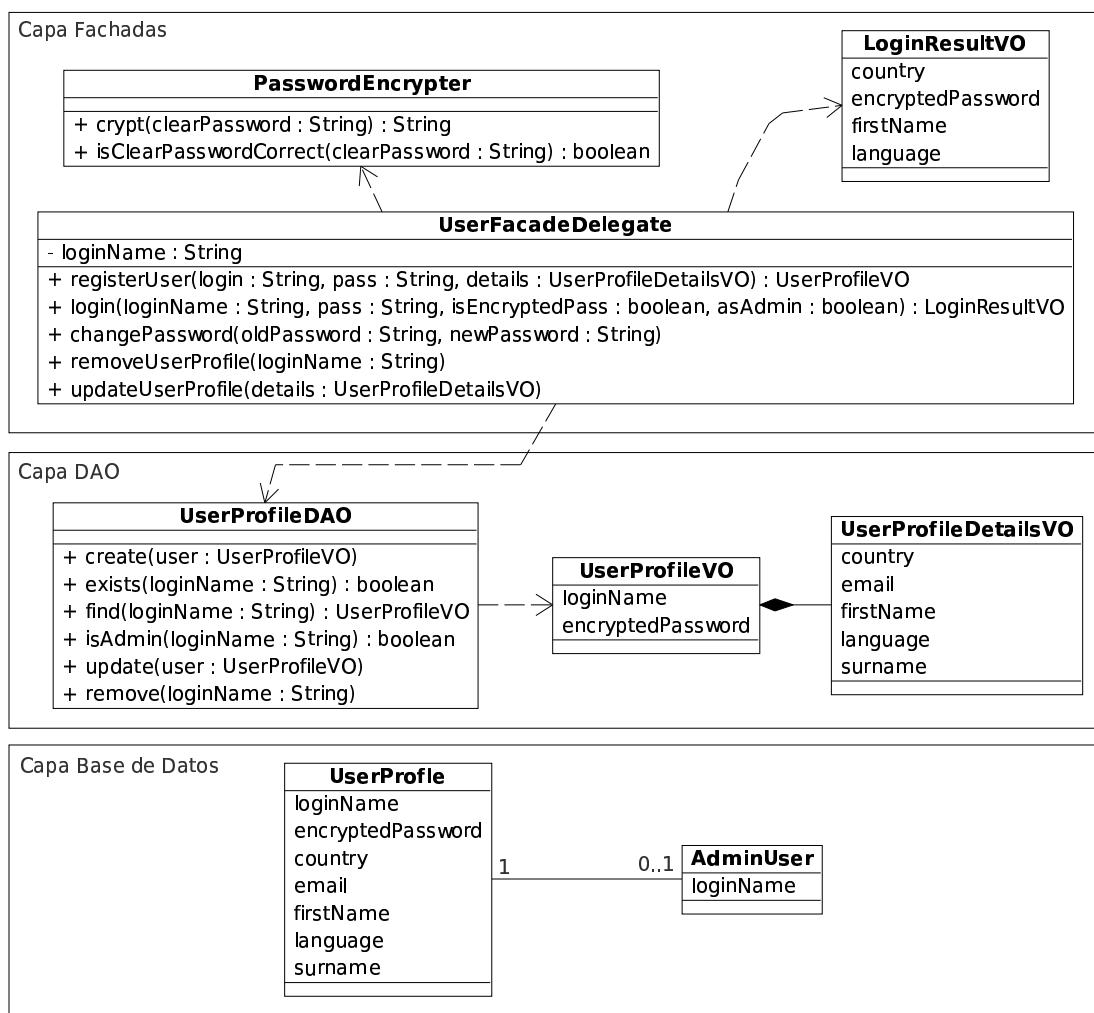


Figura 4.9: Clases del modelo que se encargan de gestionar las cuentas de usuario.

modo se pueden guardar las contraseñas encriptadas en la base de datos. También se utiliza para averiguar si alguna contraseña es correcta (si ha sido previamente encriptada con esta clase).

El método `login` sirve para identificar un usuario. Si el usuario y la contraseña son correctos, se devuelven en `LoginResultVO` algunos datos útiles para mantener la sesión de usuario a nivel controlador, en caso contrario devuelve un error (usuario inexistente, contraseña incorrecta, etc). También se utiliza para identificar administradores (con el flag `isAdmin` igual a `true`).

La representación de los usuarios se divide en dos clases, que son `UserProfileVO` y `UserProfileDetailsVO`. En la primera van los datos identificativos mínimos, y en la segunda los datos a mayores. Esta separación facilitará en un futuro la suma de nuevos atributos a los perfiles de usuario (por ejemplo el número de teléfono o el DNI) porque en algunas partes del código son necesarios y en otras solamente es necesario el usuario y la contraseña.

La representación en la base de datos solamente necesita dos tablas, una para guardar los datos de cada usuario (con la contraseña encriptada, por seguridad) y otra para decidir cuales de esos usuarios son administradores (los `loginName` incluidos en esta tabla son los administradores).

4.3.2. Guaridas

El modelo de las guaridas es bastante sencillo. Hay un DAO para crear, encontrar o modificar guaridas (no es necesario eliminarlas porque cuando se elimina un usuario ya se eliminan todos sus datos asociados en la base de datos, debido al arrastre de las claves *foráneas*). La dirección geográfica de una guarida se representa en un objeto externo (`AddressVO`). Una guarida también contiene una serie de salas, cuya estructura se explica en la próxima sección.

La fachada de las guaridas puede encontrar una guarida a través de su dirección o a través del nombre de su dueño. El método `findBuilding` devuelve un conjunto de guaridas (vecinas en la misma calle) dentro de un `BuildingChunkVO`, que no es más que un objeto con una lista de `LairVO` y unos métodos adicionales

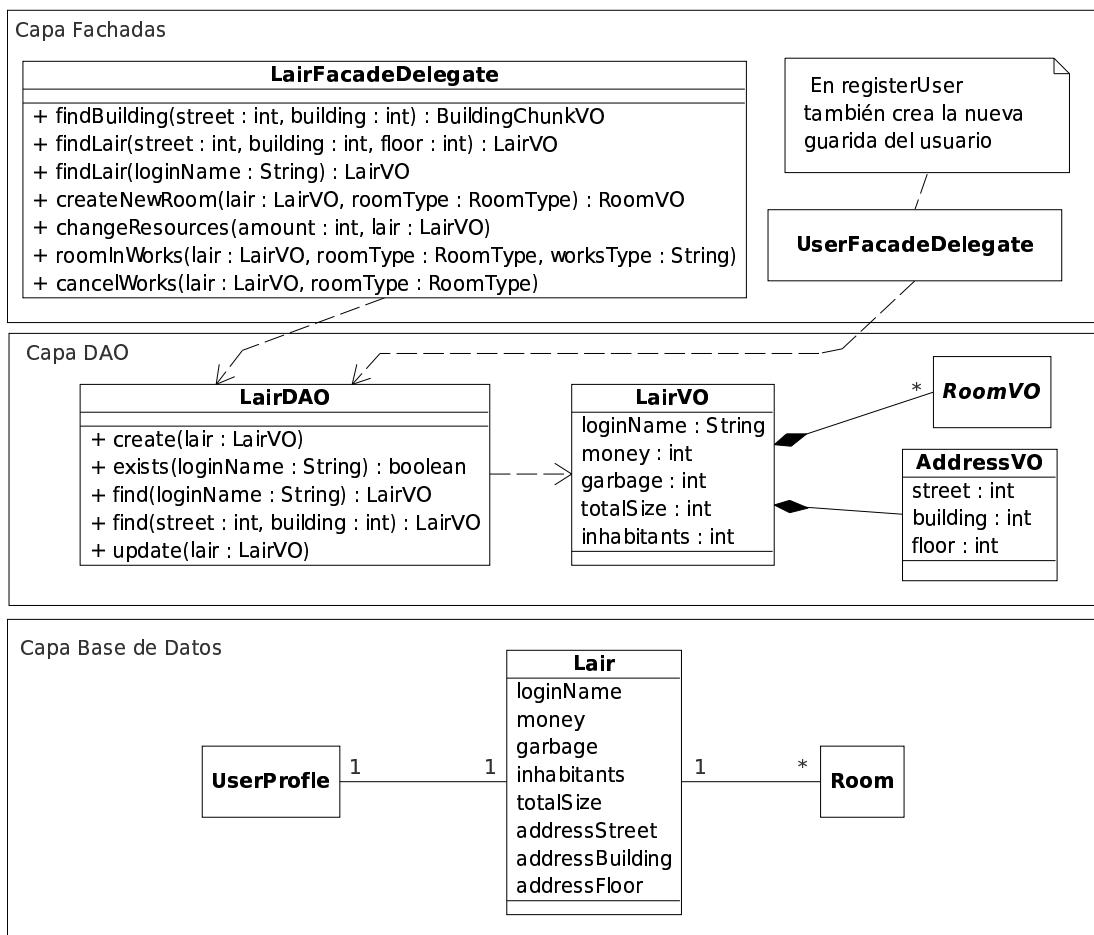


Figura 4.10: Clases del modelo relacionadas con la guarida del jugador.

para saber si hay más guardadas a la derecha o a la izquierda (siguiendo el patrón *Page by Page iterator*). Esta fachada también contiene métodos relacionados con las salas de la guarida (no hay una fachada extra para las salas, sino que los casos de uso que afectan a una única sala se ponen aquí también). Se pueden intercambiar recursos, poner una sala en obras o cancelar las obras de una sala (al cancelar obras también se utiliza el DAO de las tareas que se incluye en la figura 4.16 para anular las tareas de construcción).

La fachada de usuarios también utiliza el `LairDAO` porque cuando se registra un nuevo usuario (método `registerUserProfile`) se crea automáticamente su nueva guarida y se inicializan algunas salas, es decir, que también utilizará el `RoomDAO` que se muestra en la figura 4.11.

4.3.3. Salas

El DAO de las salas se utiliza desde las fachadas de la guarida y de usuarios, que se describen en las secciones anteriores. En el diagrama de la figura 4.11 no se incluye la capa de las fachadas porque lo importante en este caso es mostrar como se representan las salas de la guarida.

Las salas de la guarida pueden estar publicadas o no, y también pueden estar en obras o no. Como es necesario almacenar información en los estados de la sala, hay que incluir las tablas `PublishedRoom` y `RoomInWorks` en la base de datos. Para la publicación de las salas, si una sala es pública, se almacenan el precio de entrada, el precio especial para los miembros de la alianza, y el texto de información que haya fijado el usuario. Cuando la sala no es pública no se almacena nada (de este modo, el DAO ya sabe si una sala es pública o no). Con el estado de las obras sucede igual, pero además se incluye el campo `workstype` para saber que tipo de obras se están realizando. En la representación de la base de datos se incluyen tablas adicionales, y en la representación en memoria se utiliza el esquema del patrón *state*, dejando que `Room` delegue las operaciones sensibles al estado a los objetos estado de los que está compuesto.

Como la sala, sus datos de publicación y sus datos de estado en obras se

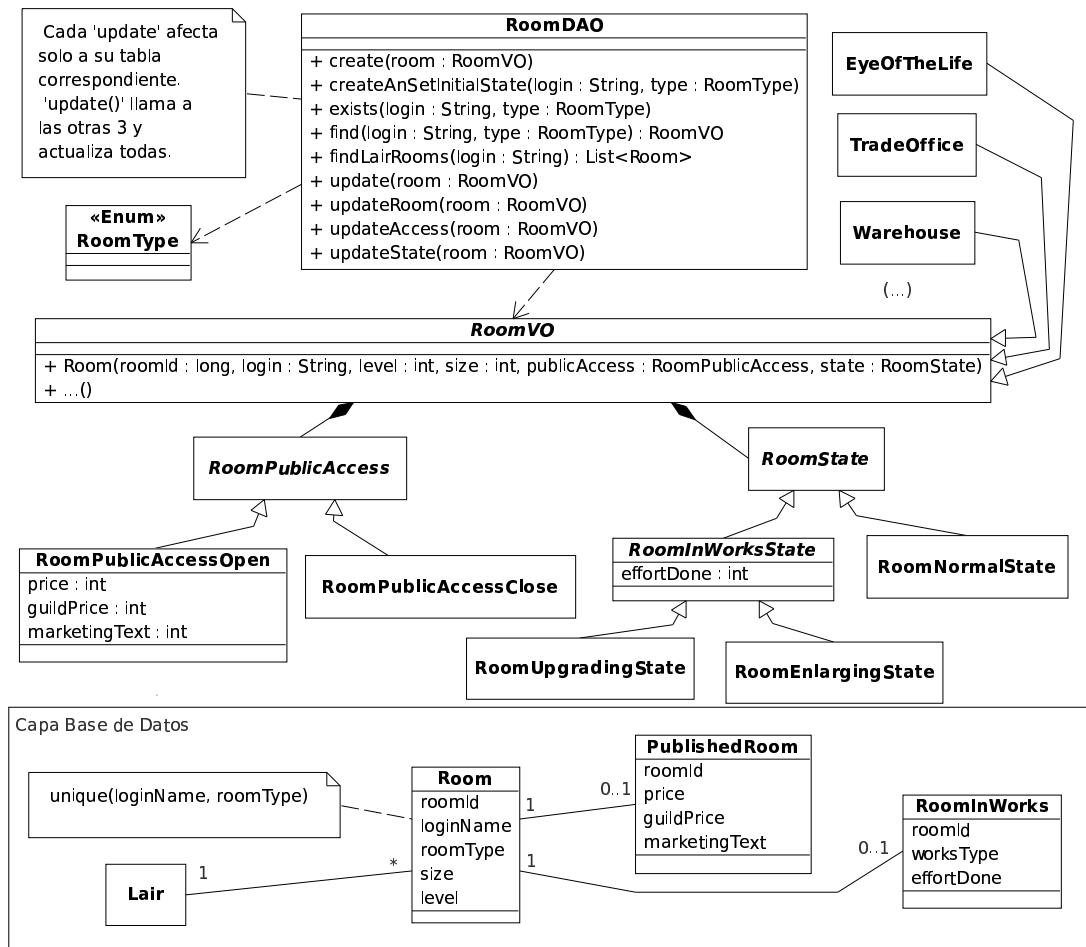


Figura 4.11: Diagrama de clases de las salas de la guarida.

guardan en tablas diferentes, entonces se exponen tres operaciones distintas para modificarlos: `updateRoom`, `updateAccess` y `updateState`, que afectan únicamente a la tabla correspondiente. Para simplificar, también se añade el método `update`, que modifica las tres tablas (este se usa en caso de duda).

Gracias al atributo `roomType`, el DAO sabe qué clase concreta de sala debe instanciar (`EyeOfTheLife`, `TradeOffice` ...). Los diferentes tipos de sala tienen distintos comportamientos, pero necesitan los mismos datos, por lo tanto se pueden guardar todas en la misma tabla.

4.3.4. Monstruos

La fachada de los monstruos tiene métodos para ver los huevos que hay en una guarida, para comprar un huevo, venderlo, incubarlo, hacer nacer un monstruo, ver los monstruos de una guarida o ver los detalles de un monstruo concreto.

Aunque en el modelo conceptual se veían los huevos como un estado más de la vida de un monstruo (figura 3.16 del capítulo de análisis), a la hora de implementarlo, resulta que es mejor tratarlo como un objeto persistente a parte (figura 4.12). Es cierto que comparten algunas características (raza de monstruo, parientes, guarida a la que pertenecen) sin embargo su comportamiento es muy distinto. Los huevos de monstruo no tienen una lista de tareas, no tienen atributos y tampoco tienen condicionantes. Simplemente tienen dos estados: sin incubar o incubado, que determinan si el huevo está listo para eclosionar.

Al haber dos objetos persistentes (`Monster` y `MonsterEggVO`) también son necesarios dos DAOs (`MonsterDAO` y `EggDAO`), pero esto en realidad simplifica la codificación, ya que el DAO de los monstruos es mucho más complejo que el de los huevos. La complejidad de la representación de los monstruos, tanto a nivel objetos como de base de datos, se puede apreciar en el diagrama de la figura 4.13, y el DAO de los monstruos es el encargado de interpretar y traducir de una representación a otra.

En el modelo conceptual se había dicho que un monstruo tiene un estado que depende de su edad (aquí omitimos el estado huevo y lo representamos en una

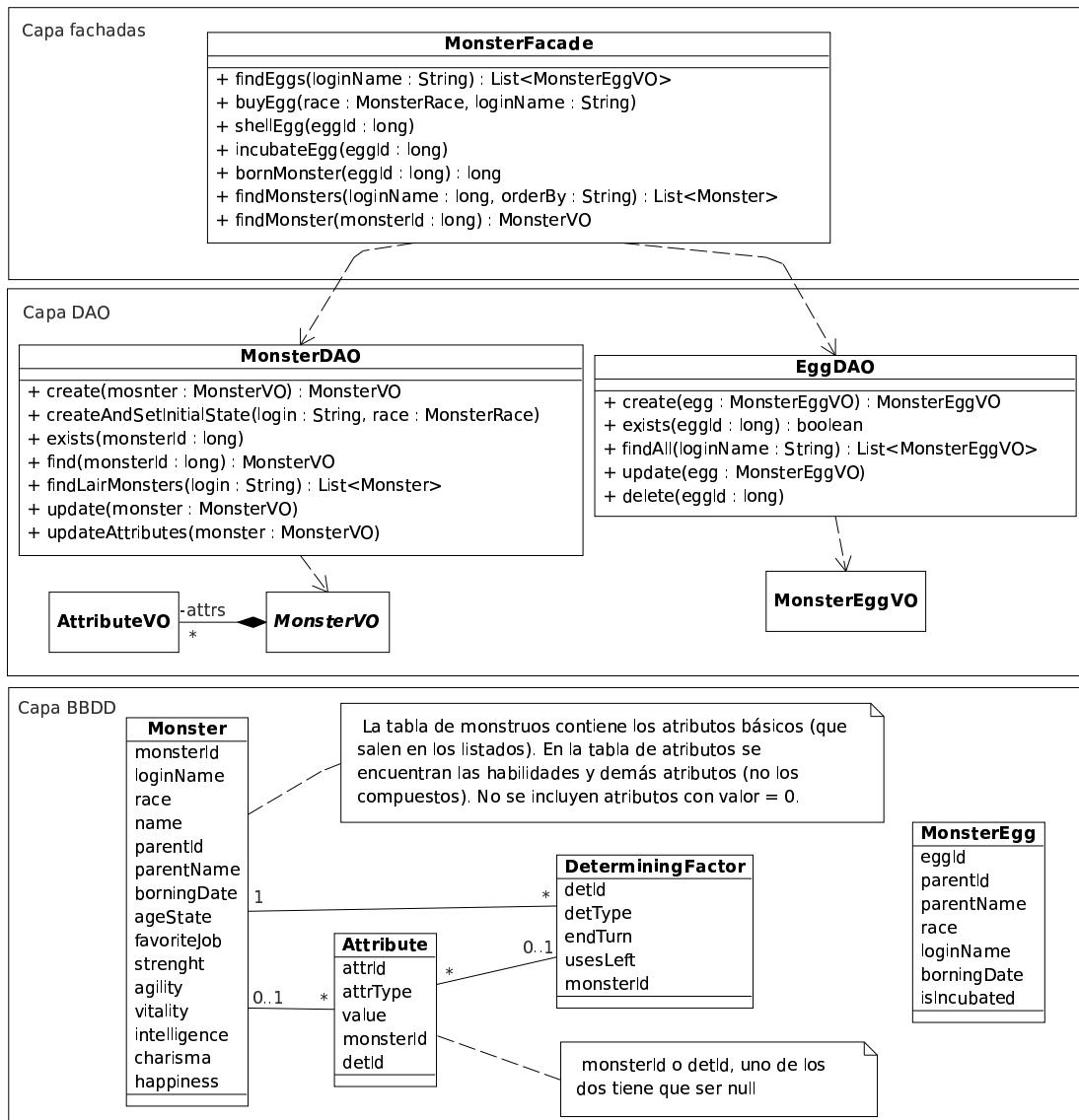


Figura 4.12: Diagrama de clases con la fachada, los DAOs y la representación en la base de datos de los monstruos y los huevos de monstruo.

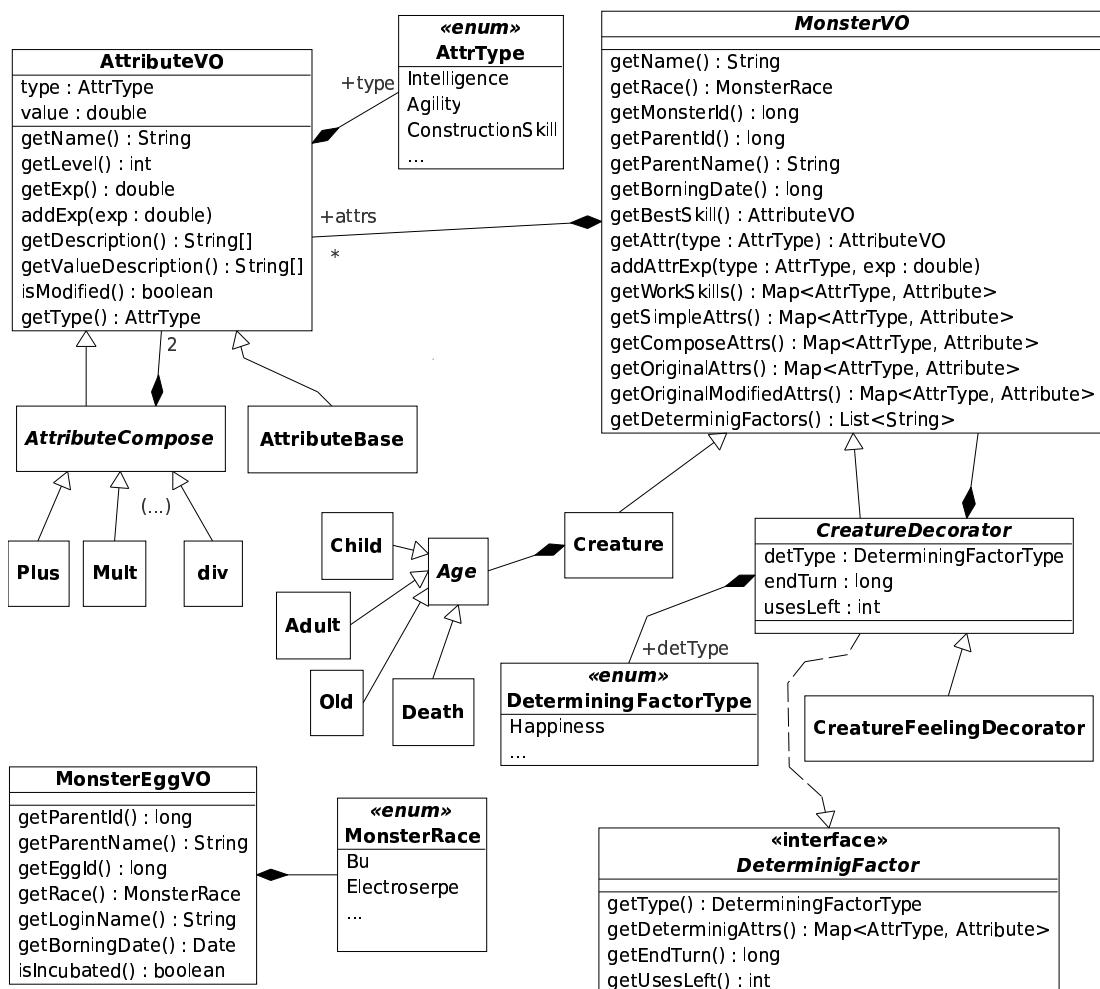


Figura 4.13: Diagrama de clases con los VOs que encapsulan los detalles de los monstruos.

clase aparte), también tiene una serie de atributos (`Monster` está compuesto de varias instancias `Attribute`, esto es igual) y una serie de condicionantes (interfaz `DeterminingFactor`), pero en este diagrama no se ve claro la composición de los condicionantes. El caso es que los condicionantes modifican el comportamiento del monstruo, por norma general añaden o quitan valor a alguno de sus atributos, así que la estructura más adecuada para este caso es la que sugiere el patrón *decorator*¹⁰. Un condicionante es entonces un `CreatureDecorator`, que es a su vez un `Monster`, y está compuesto de otro `Monster` (que es el monstruo al que está decorando). De este modo se pueden “apilar” varios decoradores sobre el mismo monstruo, donde el monstruo original está implementado por la clase `Creature`. Los atributos están en la clase `Monster`, por lo tanto los `CreatureDecorator` también tienen atributos, que sirven para determinar los atributos originales del `Creature` que son modificados (el valor del atributo del decorador puede ser positivo o negativo, indicando la cantidad de valor que se suma o se resta al atributo original). Además los `CreatureDecorator` tienen atributos propios, como son `endTurn` o `usesLeft`, que sirven para indicar cuánto tiempo o cuántos usos le quedan al condicionante. Como necesitan almacenar sus propios datos y además una serie de atributos, en la representación de la base de datos los atributos pueden pertenecer a un monstruo o a un condicionante. En la figura 4.14 se muestra un ejemplo de cómo se organizan los objetos que forman un monstruo. El cliente desde afuera solamente ve un objeto de la clase `Monster`.

La interfaz `DeterminingFactor` es necesaria porque desde la vista es posible que se quieran enumerar los condicionantes (armas, amuletos, sentimientos ...) que tiene el monstruo, y así se exponen solamente las operaciones que son necesarias.

Según el diseño conceptual, los atributos pueden ser simples o compuestos. Los atributos compuestos están formados de otros atributos (su valor depende de una fórmula que combina los valores de sus atributos agregados). Para modelar

¹⁰Un decorador (*decorator pattern*) modifica las propiedades de un objeto, siendo del mismo tipo que éste para aprovechar las ventajas de la abstracción y la herencia de clases

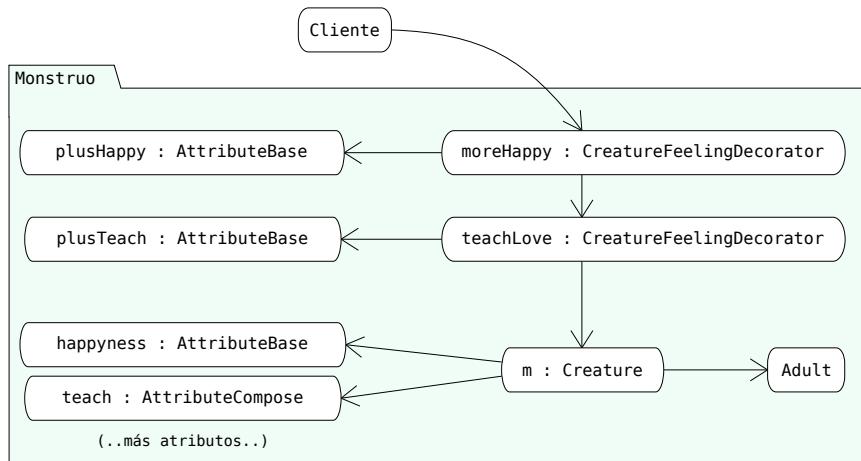


Figura 4.14: Diagrama de objetos donde hay un monstruo adulto con dos condicionantes, uno aumenta la felicidad y el otro la capacidad de enseñanza.

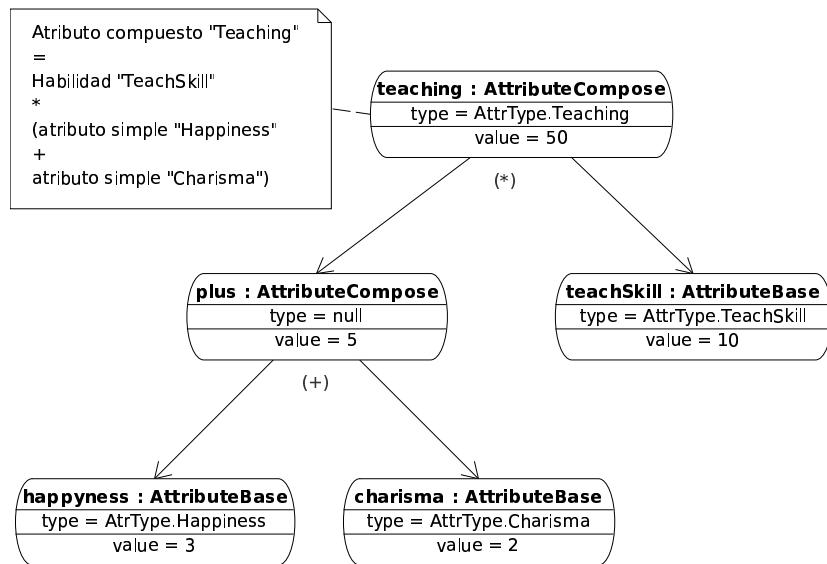


Figura 4.15: Diagrama de objetos con la representación de un atributo compuesto.

esto, se ha utilizado el patrón *composite*¹¹. La clase `AttributeCompose` es abstracta y está compuesta de dos atributos (se forma un árbol binario). Después se implementan varios tipos de `AttributeCompose` como pueden ser la suma, la multiplicación, la división, etc. Combiando varios de estas operaciones se obtiene una fórmula en base a atributos simples (las habilidades también son atributos simples). Un ejemplo se puede ver en la figura 4.15, donde el atributo compuesto para la enseñanza se forma a partir de la habilidad acumulada de enseñanza y los atributos simples `felicidad` y `carisma`. En la figura 4.25 de la sección 4.5 también se puede ver un ejemplo de cómo se reliza la operación `getDescription` sobre un `AttributeCompose`.

4.3.5. Tareas

La fachada de las tareas presenta por el momento dos sencillas operaciones: “sugerir tareas” (`getSuggestedTasks`) y “asignar tarea” (`setTask`). La primera devuelve una lista de tareas que serían posibles para asignar a un monstruo en el turno dado. La segunda asigna una tarea y la hace persistente en la base de datos. Más adelante, en la sección 4.4.3, se verá que estos dos casos de uso se van a resolver mediante *Ajax*, aunque esto no importa en absoluto para el diseño del modelo, que supondrá únicamente la parte del proceso completo que se encarga de la persistencia y de la lógica de negocio (ventaja del patrón MVC).

Cuando se intenta asignar una tarea puede que no sea posible por una serie de razones, que se incluyen en la excepción `InvalidTaskExceptions`, que tiene los métodos adecuados para enumerar las excepciones del tipo `InvalidTaskException` que contiene. La persistencia se maneja en términos de la clase `TaskDAO`.

La representación de una tarea consta de: turno (turno del día en el que se realiza la tarea), identificador del monstruo, identificador de la sala, rol (tipo de tarea) y atributo del monstruo al que va dirigido la tarea (para tareas como la

¹¹El patrón *composite* permite una composición de objetos que se puede utilizar entre otras cosas para representar estructuras en árbol, como por ejemplo una jerarquía de ficheros y directorios.

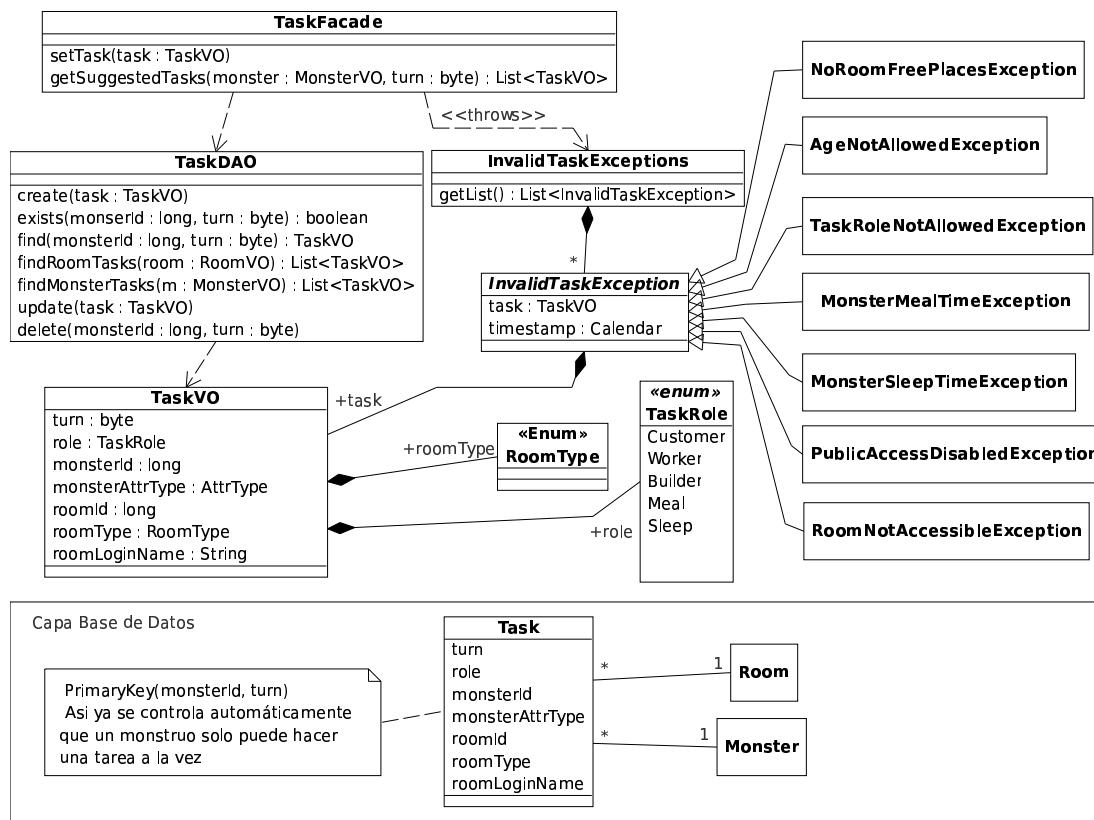


Figura 4.16: Diagrama de clases de las tareas.

enseñanza, donde se dan clases de un atributo o habilidad específicos). El resto de atributos son por eficiencia, para que cuando se muestren los datos de una tarea no haya que realizar nuevas consultas a la base de datos.

La parte complicada de las tareas es la validación, ya que hay muchas restricciones impuestas en las salas (ver sección 3.5.4 del análisis), y no se puede dejar que haya tareas incorrectas en la base de datos. De ahí que en el diagrama de la figura 4.16 se incluyan los errores que puede tener una tarea (en los diagramas anteriores no se incluyen los errores por simplicidad).

En la clase `Monster` del diagrama con los detalles de los monstruos de la figura 4.13 no se incluyen las tareas, pero realmente en esa clase hay una serie de métodos para ver las tareas que tiene un monstruo (ver todas las tareas, ver tarea realizada en un turno, etc). Lo mismo pasa con la clase `Room` del diagrama de la figura 4.11. Tanto los monstruos como las salas tienen una lista de tareas, que se puede obtener utilizando el `TaskDAO`.

4.3.6. Configuración

En la aplicación *Thearsmonsters* hay algunos parámetros que necesitan ser configurados para su despliegue. La configuración se lleva a cabo dando ciertos valores a estos parámetros de forma accesible, en un archivo de configuración o mediante recursos *JNDI*. Este concepto no estaba contemplado en la fase de análisis porque en realidad está más ligado a la implementación que al concepto del juego. Sin embargo a nivel de diseño es importante clarificar cómo se aborda esta necesidad.

Como se puede ver en la figura 4.17, la clase principal para la configuración es `ConfigurationParametersManager` (que está en el paquete *j2ee.util.configuration* del subsistema *standardutil*). Esta clase trata de inicializar los parámetros en un bloque estático (el bloque estático se ejecuta cuando se carga la clase, es decir, al reiniciar la aplicación), guardándolos en memoria (en una tabla *Hash*) y haciéndolos accesibles con el método `getParameter`. Al inicializar los parámetros se comprueba si se puede utilizar JNDI o hay que leer los parámetros de un

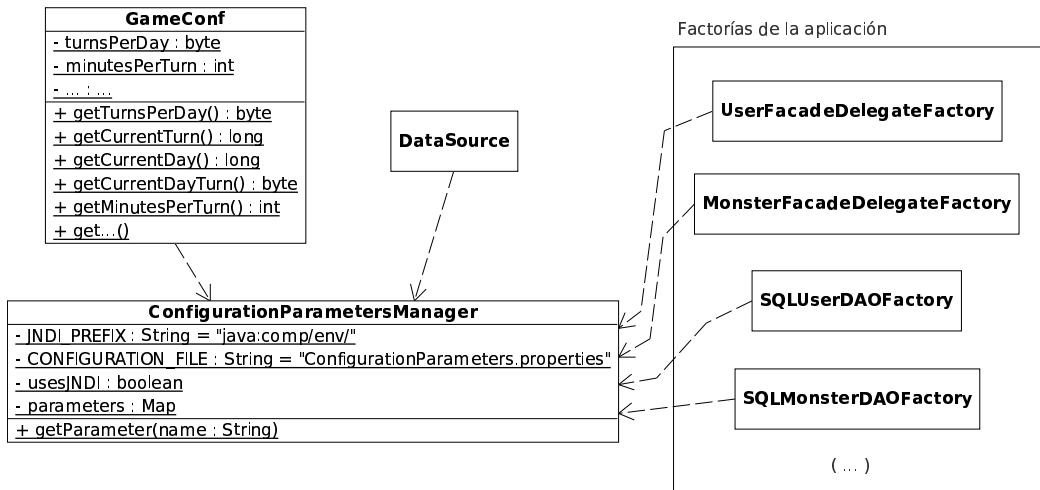


Figura 4.17: Estructura de clases que organiza los aspectos relacionados con la configuración.

fichero (en general, se utiliza el fichero para validación y uso local, y JNDI en producción). Esta clase la utilizan las factorías (para saber qué clases concretas deben instanciar), los objetos **DataSource**, que leen en la configuración cómo deben conectarse a la base de datos (url, usuario, contraseña ...) y todas las clases que necesiten leer algún parámetro de configuración.

Como también hay parámetros de configuración de los que depende la lógica del juego (no sólo se guardan parámetros para conectarse a la base de datos), se incluye una clase adicional que simplifica la lectura de parámetros relacionados con el juego llamada **GameConf** (figura 4.17). El propio juego también se puede configurar, indicando los minutos que tiene un turno de juego (por defecto 60 = una hora), cuántos turnos hay en un día de juego (por defecto 24), en qué fecha se realizó el primer turno del juego, cuántas guardadas puede haber como máximo en la base de datos, etc. Conociendo estos parámetros también se pueden calcular muchos otros (por ejemplo, con la fecha del turno inicial, la duración de un turno y la fecha actual se puede calcular el turno actual), por eso se incluyen métodos que realizan los cálculos de forma simplificada. Esta clase inicializa sus atributos en un bloque estático, leyéndolos de **ConfigurationParametersManager**.

Todas las acciones del juego que necesiten saber datos relacionados con la configuración del juego deberán utilizar `GameConf`.

4.4. Controlador y Vista

En el apéndice A se exponen los motivos por los cuales el *framework* seleccionado para implementar la capa Web en *Thearsmonsters* es *Struts*. También se comentan las otras alternativas que fueron rechazadas: *Ruby on Rails*, *Django*, *Tapestry*, *Microsoft .Net*, etc.

En las aplicaciones Web de gran tamaño (como *Thearsmonsters*) presentan una serie de problemas comunes. Por ejemplo, si necesita utilizar sesiones de manera robusta tiene que aplicar *URL rewriting* a todas las URLs que genera o hace una redirección, y necesita hacerlo de forma sencilla y directa. También es preciso hacer sencilla la implementación de formularios, control de errores, redirección, seguridad, etc. Para procesar las peticiones *HTTP* y construir una respuesta (HTML, WML, XML ...) hacen falta mecanismos que mantengan separado el código de la vista y del controlador, que sean simples y manejables, y un buen método para conseguirlo es utilizar el patrón *Front Controller* (figura 4.18), que unifica el procesado de las peticiones (*doGet*, *doPost*, etc.) en el controlador y delega las partes diferenciadas en acciones concretas, que son mucho más sencillas de implementar. [23]

Además también hay problemas al escribir texto en HTML porque hay muchos caracteres reservados, problemas para internacionalizar la aplicación, para reutilizar partes de la vista, etc.

Por lo tanto es necesario utilizar un *framework* que proporcione al menos una buena solución a todos estos problemas, dándole el siguiente aspecto a la aplicación Web:

- **Modelo**

- Conjunto de clases que implementan la lógica de negocio y la persistencia de datos
-

- Independientes de la vista
- **Controlador**
 - Servlet *FrontController* y clases acción
 - Desacopla la vista del modelo
- **Vista**
 - Páginas cuidadas con *tags* sencillos, que puedan utilizar los diseñadores
 - Usan muchos *custom tags* reutilizables (la mayor parte proporcionados por el *framework*)
 - No es necesario utilizar lenguajes de programación complicados

Utilizar *Struts* es una solución bastante satisfactoria a los problemas anteriormente mencionados. En los siguientes apartados se aborda el diseño del controlador y de la vista bajo el enfoque práctico de este *framework*.

4.4.1. Estructura condicionada a JSTL y Apache Struts

Para que esta sección no sea demasiado extensa se va a suponer que el lector ya cuenta con los conocimientos necesarios para entender el texto (ver entradas de la bibliografía [4], [31], [23] y [21]). Lo que se pretende mostrar en este apartado es cómo se han estructurado los paquetes del controlador y la vista en base a las restricciones que presenta el uso de *Struts*.

Los diferentes paquetes se dividen en bloques lógicamente relacionados, haciendo la conveniente separación entre modelo, vista y controlador. La estructura de paquetes en esta aplicación es la siguiente:

- **es.udc.mizquierdo.j2ee.thearsmonsters**: Contiene los paquetes específicos del juego
 - **http** Conjunto de paquetes para la vista y el controlador

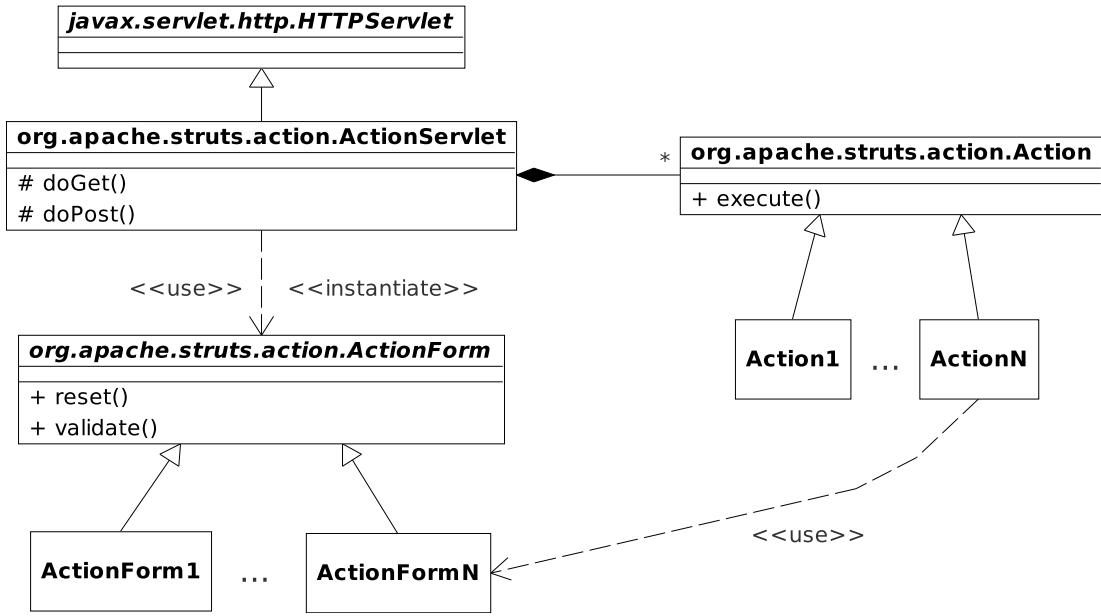


Figura 4.18: El patrón *Front Controller* en *Struts*.

- controller
 - actions: Acciones que necesitan lógica específica y son instanciadas por el Front Controller
 - front controller: Clases que implementan el Front Controller (figura 4.18). Se encargan de recibir las peticiones HTTP y utilizar la acción correspondiente, pasándole los parámetros y controlando el flujo de la aplicación.
 - view
 - actionforms: Formularios en forma de propiedades, con validación.
 - applicationobjects: Objetos como listas de elementos que son estáticos en la aplicación.
 - messages: Todo el texto que salga en la vista está internacionalizado.

- **model**: Conjunto de paquetes para el modelo
- **es.udc.mizquierdo.j2ee.util**: Utilidades comunes en una aplicación web *Struts*
 - **configuration**: Clases para guardar parámetros de configuración.
 - **sql**: *DataSource* que conecta con la base de datos, acciones por defecto, control de transacciones, etc.
 - **exceptions**: Errores comunes a una aplicación web, como *InternalErrorException* o *ModelException*.
 - **struts**: Acciones, formularios, validación de formularios y componentes de la vista por defecto.

Las diferentes vistas se dividen en subcarpetas dentro del directorio *WEB-INF*¹². La forma de componerlas es a base de plantillas, ya que en la vista del juego, la mayoría de pantallas tienen la misma estructura que se muestra en la figura 4.19. Otras tienen la estructura de la portada, otras de la administración, pero en general son pocos los tipos de *layout* que hay. Para estructurar los componentes de las vistas se utiliza el *plugin* de *Struts* llamado *Tiles*, que utiliza un caso particular del patrón *Composite View* (*Core J2EE Patterns*). Gracias a ello, utilizando un simple fichero de configuración XML, se definen los componentes que se repiten y los que hay que sustituir en cada vista. Así los distintos ficheros *.jspx* contienen tan solo el contenido dentro del *layout*, evitando tener que repetir código.

4.4.2. SessionManager

Anteriormente se ha comentado que una parte fundamental de una aplicación como *Thearsmonsters* es mantener la sesión del usuario, es decir, que la aplicación

¹²En una aplicación Web *Java EE* el directorio WEB-INF incluye las carpetas y los ficheros que son accesibles desde el navegador

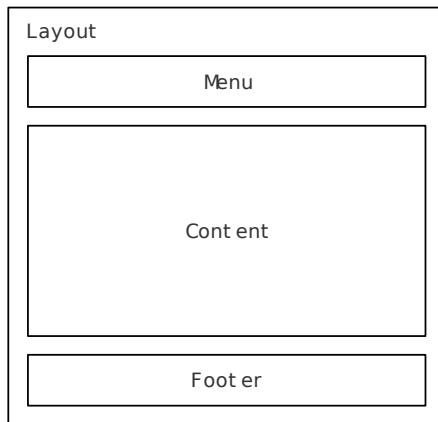


Figura 4.19: *Layout* genérico de las pantallas durante el juego.

sepa en cada momento quien está accediendo a ella para permitirle el acceso solamente a los lugares adecuados.

Las aplicaciones Web se acceden con el protocolo *HTTP*, que no tiene estado. Cada petición es independiente de la anterior por lo que para saber de qué usuario se trata hay que añadirle la información necesaria para que la aplicación pueda identificar la sesión.

Entonces ¿Cómo crea y mantiene una sesión en una aplicación Web? Este es un problema genérico, que se soluciona entre la propia especificación de *Servlets Java EE* y el *framework Struts*. Básicamente, lo que se hace es comprobar en cada petición *HTTP* si tiene algún código de sesión asociado. Si no tiene un código, entonces se crea una nueva sesión (se supone que es una nueva conexión con la aplicación). En el servidor se mantienen varias sesiones activas, identificadas por su propio código (el código debe ser largo y con muchos caracteres diferentes para que no pueda ser acertado por casualidad), que caducan al cabo de un tiempo (por norma general 30 minutos). Después de procesar la petición, en la respuesta del servidor al navegador, se le indica cual es su código de sesión, y en las próximas conexiones el navegador envía el código para que el servidor lo reconozca. La forma más cómoda de mantener el código de sesión en el navegador es mediante las

*cookies*¹³, así cada vez que se recibe una petición se lee de la cookie de sesión cual es el código asociado. Lo malo es que no todos los navegadores tienen soporte para *cookies*, o pueden estar desactivadas, o pueden borrarse en cualquier momento. Cuando no se pueden utilizar las *cookies* se usa la técnica *URL rewriting*, que consiste en asignar a cada enlace generado en la vista un parámetro adicional (llamado `session_id` o similar) que contenga el identificador de la sesión, lo cual es menos elegante y menos seguro que las *cookies* pero funciona para todos los casos. [23]

En Struts se mantiene la sesión de forma casi transparente para el programador. Y gracias a los tags `html:link` se asegura el *URL rewriting* en los enlaces, en caso de que no funcionen las *cookies*. Sin embargo con esto solamente conseguimos identificar una conexión para asociarla a su sesión correspondiente. También es necesario identificar al *usuario* (el mismo usuario se puede conectar cada vez desde una máquina diferente, o en la misma máquina conectarse varios usuarios). Por lo tanto es necesario crear los casos de uso “registrar usuario”, “identificar usuario” y “cerrar sesión”. Estos casos de uso trabajan en varias capas de la aplicación: a nivel base de datos hay que guardar los datos de los usuarios registrados, a nivel modelo hay que reconocer si un usuario está registrado y devolver los errores asociados en caso de que no lo esté, y a nivel controlador hay que reconocer y mantener al usuario identificado, además de seleccionar las vistas adecuadas dependiendo de si es un usuario registrado o no y de poner la información necesaria en la sesión.

Además se añade la funcionalidad de “recordar al usuario en una máquina concreta”. Mediante el uso de *cookies* se puede guardar en un navegador concreto el nombre de usuario y su contraseña (cifrada) para que cuando se vuelva a conectar, aunque su sesión haya caducado, se pueda identificar y se cree una nueva sesión para el mismo.

Para abstraer todas estas funcionalidades en el controlador se crea la clase

¹³Las *cookies* del navegador son datos asociados a una dirección web, y cada vez que se conecta a esa dirección se envían todas las *cookies* al servidor. De esta forma los servidores pueden almacenar datos en los navegadores cliente.

SessionManager
<pre> - COOKIES_TIME_TO_LIVE_REMEMBER_NY_PASSWORD : int - COOKIES_TIME_TO_LIVE_REMOVE : int - ENCRYPTED_PASSWORD_COOKIE : String - LAIR_SESSIOIN_ATTRIBUTE : String - ... : ... + changePassword(request : HttpServletRequest, response : HttpServletResponse, oldPassword : String, newPassword : String) + getMyLair(request : HttpServletRequest) : LairVO + isUserAutenticated(request : HttpServletRequest) : boolean + login(request : HttpServletRequest, response : HttpServletResponse, login : String, pass : String, isEncryptedPass : boolean) : LoginResultVO + registerUser(request : HttpServletRequest, login : String, pass : String, details : UserProfileDetailsVO) + ...() </pre>

Figura 4.20: Méodos representativos de la clase **SessionManager**.

SessionManager (figura 4.20), que sirve para encapsular la mayoría de operaciones que tiene que hacer la aplicación con la sesión. Los métodos *login* y *register* se encargan de identificar o de registrar al usuario respectivamente, utilizando la fachada del modelo **UserFacadeDelegate** para recuperar o registrar sus datos, guardarlos en la sesión, crear las *cookies* si es necesario y devolverlos al controlador de forma transparente. También es importante el método *loginFromCookies*, que se utiliza en un filtro que se aplica en todas las peticiones para ver si hay *cookies* con el nombre de usuario y contraseña y así poder identificar al usuario de manera automática.

Las acciones que sean susceptibles de modificar datos en la sesión deben ir en esta clase. Por ejemplo *updateUserData* puede modificar el *locale* (idioma seleccionado) en la sesión. Por último, maneja los datos guardados en la sesión a modo caché, como la guarida del usuario. Así desde el código del controlador, si queremos recuperar la guarida del usuario solamente hay que hacer **SessionManager.getMyLair(request)**, y ese método se encarga de comprobar si la guarida está en la sesión, si no está la recupera de la base de datos, la *cachea* en la sesión y la devuelve.

4.4.3. Acciones por defecto

Hay una acción para el *Front Controller* por defecto llamada **DefaultAction** la cual sirve como base para implementar cualquier otra acción. **DefaultAction** se encarga de ejecutar la acción correspondiente y de comprobar si hay errores

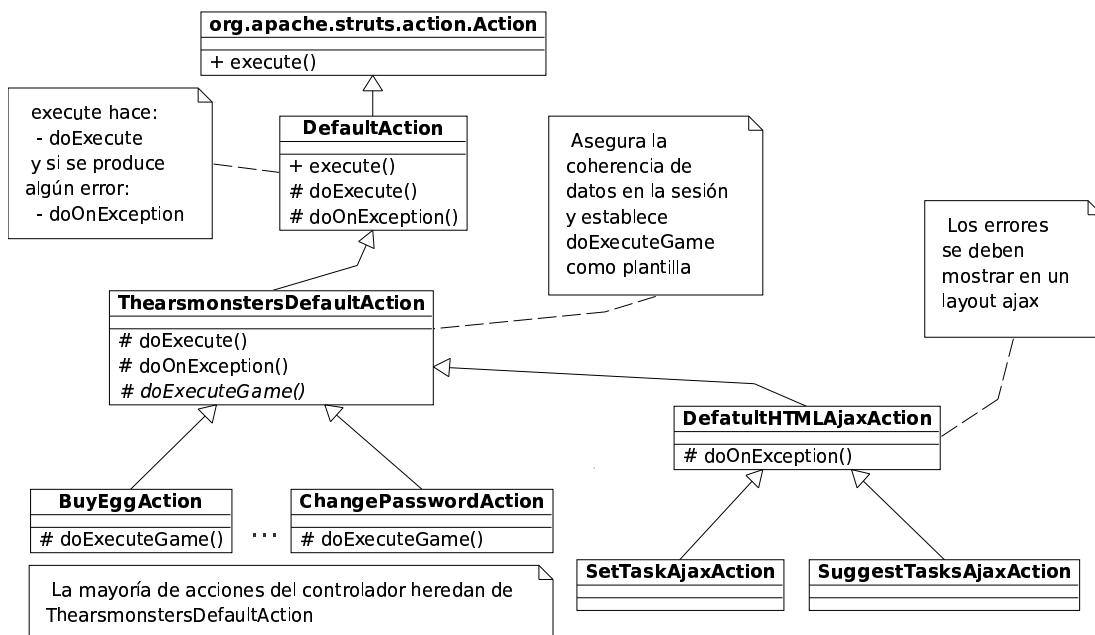


Figura 4.21: Gerarquía de acciones en el controlador de la aplicación.

para, en ese caso, redirigir a una pantalla común de error y de mostrar el error en los ficheros de *log* correspondientes.

En *Thearsmonsters* no se utiliza directamente la acción por defecto, sino que se extiende en dos ramas. Una se llama `ThearsmonstersDefaultAction`, que además de comprobar los errores y hacer *log*, se asegura de que los datos cacheados en la sesión son consistentes (invalida la caché en caso de error, y la recarga cuando sea necesario). La otra acción por defecto se llama `DefaultAjaxAction`, y en lugar de redirigir la salida a una pantalla de error, muestra un mensaje de error predeterminado dentro del marco del resultado de una acción *Ajax* (en la sección 4.7 se explica como se llevan a cabo las acciones *Ajax*).

Estas acciones por defecto utilizan las diferentes acciones que hay a modo de *plantillas* (según el patrón de diseño *template*), por lo tanto, cada acción concreta del controlador solo necesita extender un método plantilla con su propio comportamiento.

4.4.4. Filtros Incluidos

El *Front Controller* de *Struts*, `ActionServlet`, delega el procesamiento de peticiones en la clase `RequestProcessor`, cuyo método principal es `process`, que actúa como un método plantilla que se implementa en términos de otros. En particular, llama a `processActionPerform`, que ejecuta la acción que recibe como parámetro y devuelve el `ActionForward` que retorna el método `execute` de la acción.

En *Thearsmonsters* se ha aplicado el patrón *Chain of Responsibility*¹⁴ para el preprocesado de peticiones (proporcionando nuestro propio procesador llamado `ThearsmonstersRequestProcessor`). Todas las peticiones pasan por la cadena de filtros antes de ejecutarse.

Como puede verse en la figura 4.22, el `ThearsmonstersRequestProcessor` extiende a `TilesRequestProcessor` (porque esta aplicación utiliza *Tiles*), redefiniendo `processActionPerform` para ejecutar el primer filtro de la cadena (que invocará al siguiente, y así sucesivamente). Si la cadena no devuelve un `ActionForward`, continúa con el procesamiento normal de la acción (utilizando el método `processActionPerform`), en otro caso devuelve el `ActionForward` y la acción no se ejecuta.

Todos los filtros extienden de `PreProcessingFilter`. El método `process` invoca al método abstracto `doProcess`, y si éste no retorna un `ActionForward` y hay un filtro siguiente en la cadena, invoca a su método `process` (continúa con la ejecución en la cadena). Cada filtro realiza su procesamiento en `doProcess` (si se desea interrumpir la ejecución de la cadena, retorna un `ActionForward`).

Se proporcionan tres filtros de procesamiento:

- **SessionPreProcessingFilter**: mantiene la sesión por si hay que autenticarse a partir de las *cookies*. Nunca detiene la ejecución de la cadena.
- **AuthenticationPreProcessingFilter**: comprueba si el usuario está aut-

¹⁴También conocido como *Cadena de responsabilidad*, el cual establece que una serie de objetos pueden ir delegando una operación a modo de cadena

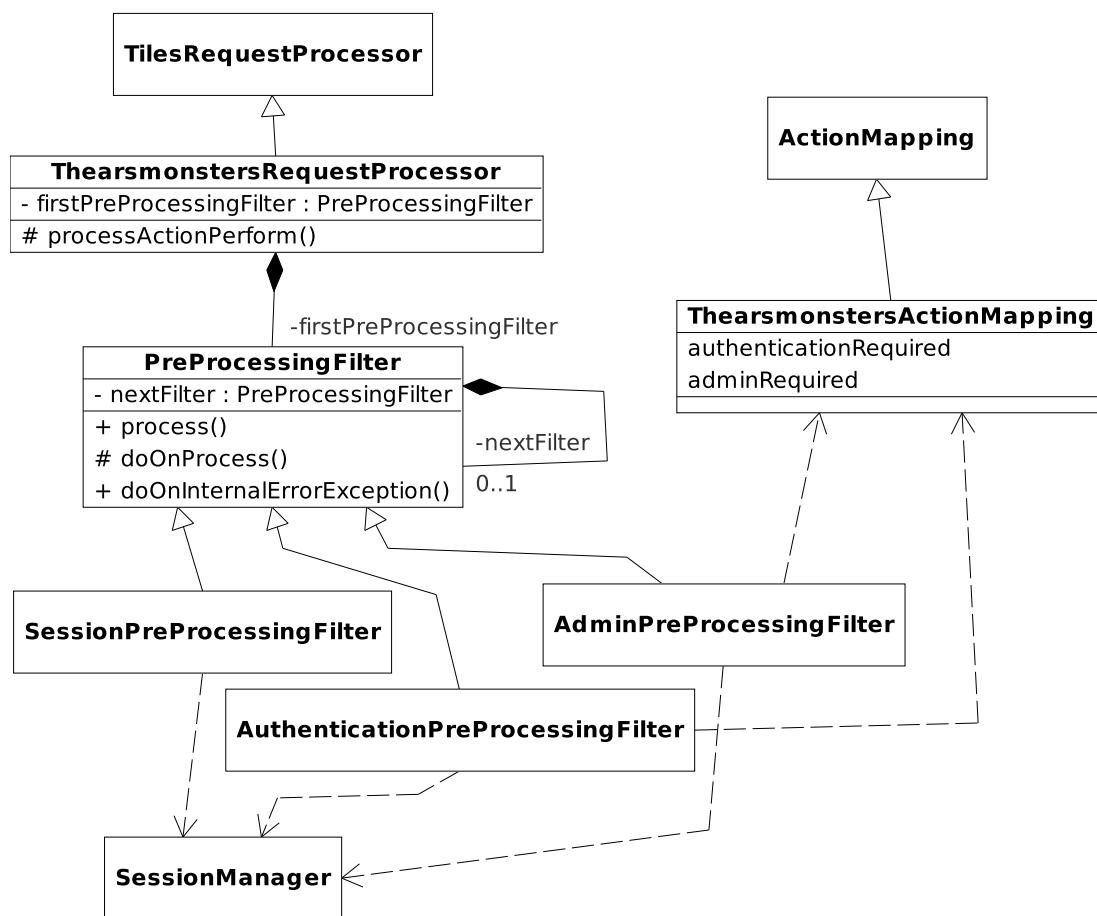


Figura 4.22: Filtros de preprocesamiento definidos en *Thearsmonsters*.

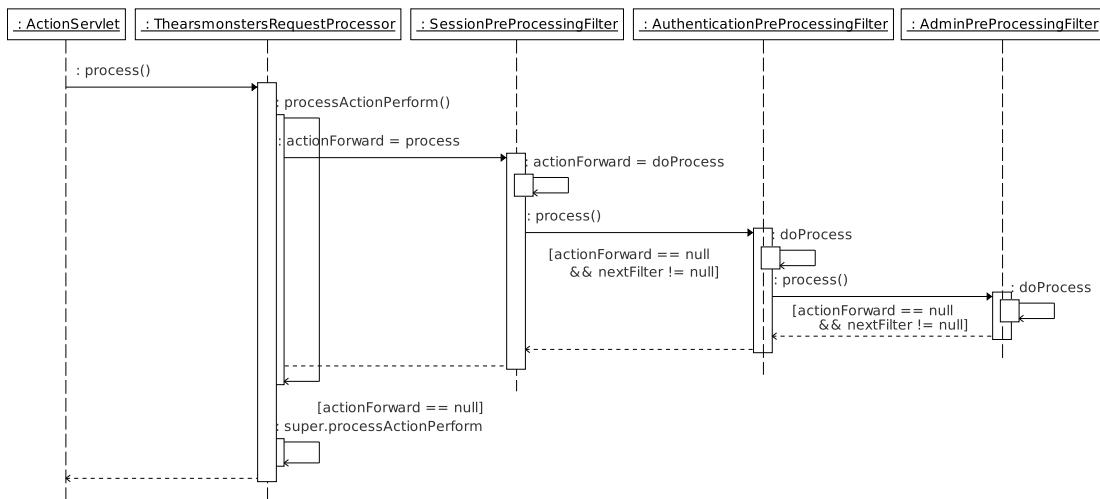


Figura 4.23: Ejecución de una petición a través de los filtros de preprocesamiento.

enticado para las acciones que lo requieren. Puede parar la ejecución de la cadena devolviendo un `ActionForward` a la página de autenticación.

- **AdminPreProcessingFilter:** comprueba si el usuario es administrador para las acciones que lo requieren. Puede parar la ejecución de la cadena devolviendo un `ActionForward` a la página de autenticación de administradores.

El orden es importante, ya que `SessionPreProcessingFilter` tiene que ejecutarse siempre de primero.

`AuthenticationPreProcessingFilter` y `AdminPreProcessingFilter` pueden comprobar en `ThearsmonstersActionMapping` si el acceso está restringido porque establece sus propiedades según se hayan definido en el fichero de configuración de *Struts*.

4.4.5. *Custom Tags* para simplificar la vista

La tecnología *JavaServer Pages Standard Tag Library* (JSTL) es un componente de *Java EE*. Extiende las páginas *JSP* proporcionando cuatro librerías de

etiquetas (*Tag Libraries*) con utilidades ampliamente utilizadas en el desarrollo de páginas web dinámicas [31]. Estas librerías de etiquetas extienden de la especificación de JSP (la cual a su vez extiende de la especificación de *Servlet*). Su API permite además desarrollar bibliotecas de etiquetas propias (los llamados *custom tags* o *etiquetas personalizadas*), permitiendo aplicar casos concretos del patrón *View Helper*¹⁵ [14].

La vista de esta aplicación es especialmente compleja debido al diseño gráfico avanzado que exige la producción de casi cualquier videojuego. Sin el uso de *custom tags* sería inviable e inmantenible el código creado a partir del diseño gráfico propuesto (que se puede ver en el capítulo 5).

Se utilizan *custom tags* por ejemplo para mostrar iconos, para crear ventanas, para hacer listas de características, para hacer botones, etc. Cada componente de la vista que se repita varias veces es susceptible a ser implementado con un *custom tag*, ya que así se favorece la refactorización del código.

Para que se entienda su funcionamiento e importancia, se muestra un ejemplo de como se hacen las ventanas que componen la mayoría de vistas en el juego, que se implementan con el *custom tag* `window.tagx`.

Según el diseño gráfico propuesto, hay que implementar unas ventanas como las de la figura 4.24, y hay que hacerlo con *HTML*, *CSS* y *JavaScript*¹⁶.

Este diseño es visualmente atractivo y otorga a la interfaz un aspecto muy limpio, intuitivo y profesional, pero a la hora de ser implementado presenta varios problemas:

- **Bordes redondeados:** Hay muchas maneras de crear bordes redondeados en una ventana *HTML*, pero siempre acarrean complicaciones. Existen cientos de artículos y soluciones propuestas para este problema (es una cuestión

¹⁵ *View Helper* son como las funciones en un lenguaje de programación tradicional, que permiten reutilizar código. Aplicadas a un lenguaje de etiquetas como *HTML* se trata de etiquetas que puedan tener sus propios atributos que afecten a la ejecución del código generado.

¹⁶ Para implementar la vista de una aplicación Web, los lenguajes que se utilizan son *HTML*, *CSS* y *JavaScript*, aunque se oculten detrás de algún *framework* o lenguaje específico como es en este caso *JSP*. También se pueden hacer con *Flash*, pero esa tecnología no se utiliza en esta aplicación.



Figura 4.24: Una ventana típica de la interfaz del juego, tiene pestañas que muestran diferentes contenidos dentro de la ventana.

bastante recurrente en los foros y páginas dedicadas al diseño Web). Generalmente el añadir bordes redondeados implica una estructura más compleja en el código HTML.

- **Pestañas dinámicas:** Cuando hay más de una pestaña en la ventana, debe ocultarse la vista de las pestañas que están inactivas y mostrar solamente el contenido de la pestaña activa. Para que esto se pueda hacer hay que utilizar código *JavaScript*, que suele ser una fuente interminable de errores. Además si los contenidos de las pestañas inactivas son muy grandes puede tener que ser necesario realizar una carga dinámica de los datos según vayan siendo necesarios (bien redirigiendo a otras páginas o bien mediante el uso de técnicas *Ajax* [28], en cualquier caso el código necesario puede llegar a ser bastante complejo).
- **Relieve de los bordes:** Para que los bordes puedan tener brillo y relieve es necesario componer la vista con múltiples imágenes (son limitaciones de la maquetación con HTML y CSS). Esto supone que hay que dividir la ventana en más componentes HTML y aplicarles el *background* correspondiente a cada uno de ellos.

- **Color con gradientes:** Los gradientes deben ser dibujados igual que el relieve en los bordes. Esto supone que la maquetación de la ventana necesita aun más elementos para ser compuesta, llegando a constar de más de 15 piezas diferentes.
- **Diferentes colores según la sección del menú:** En el diseño gráfico también se especifica que los colores de las ventanas serán diferentes dependiendo de si estamos en el menú de la guarida, de los monstruos, viendo mensajes o realizando misiones (esto ayuda al usuario a saber donde se encuentra en cada momento). Cada color que pueda tener la ventana supone un conjunto de *backgrounds* diferente (como hay 17 piezas y 6 colores diferentes, son necesarias 102 imágenes distintas), que además hay que enlazar adecuadamente para que solamente aparezcan aquellas que sea necesario según la situación.
- **Marca en la pestaña seleccionada:** Aunque parezca extraño, el saliente triangular que marca de forma elegante a la pestaña seleccionada supone una gran complicación a la hora de ser implementado debido a que la maquetación HTML no deja superponer etiquetas encima de las demás¹⁷. Para solucionar este problema hay que complicar aun más el código.
- **Diferentes tamaños:** Si las ventanas fuesen siempre del mismo tamaño sería todo más fácil (se podrían crear con unas pocas imágenes de *background* estáticas), sin embargo la vista puede componerse de varias de estas ventanas, cada una de diferente anchura y altura, por lo tanto debe poder ser reescalada según la necesidad de cada momento.
- **Internacionalización:** Este es un problema general de la vista de todo el proyecto (se trata más adelante en la sección 4.5), ya que los textos

¹⁷Esto en realidad es falso, ya que se puede aplicar la propiedad `position: absolute`, sin embargo esta solución supone un gran número de problemas, empezando porque al reescalar el tamaño de la ventana donde está el navegador, el elemento superpuesto no se mueve del sitio.

mostrados deben estar en el idioma que haya seleccionado el jugador. Esto también añadirá más complejidad a la hora de implementar la ventana.

Con todos estos problemas resulta evidente que no será fácil implementar estas ventanas (seguramente a los programadores no les parecen tan bonitas). Sin embargo tarde o temprano estos problemas pueden ser solucionados. El verdadero problema está en la “replicación de código”. Por ejemplo que para mostrar cada ventana de estas en la vista son necesarias aproximadamente 300 líneas de código, (cuando un *DIV* normal y corriente solamente supone 2 líneas de código: una para abrirlo y otra para cerrarlo). Entonces, cada vez que haya que mostrar una ventana en la interfaz ¿se copian-pegan estas 300 líneas?, ¿qué pasa cuando se detecten errores en las ventanas?, ¿habrá que arreglarlo en cada una de las ventanas que se han puesto? Además ¿quién es capaz de leer una vista donde, si se incluyen tres simples ventanas, tiene que haber como mínimo 900 líneas de código?

La solución a este “verdadero problema” se encuentra en los *custom tags*, porque permiten abstraer la implementación de estas ventanas en un único *tag*. Así que aunque la implementación de la ventana ocupe 300 líneas de código, cada vez que haya que incluir una ventana en la interfaz solamente serán necesarias 2 líneas adicionales: una para abrir el *tag*, y otra para cerrarlo.

El propio *framework Struts* (así como muchos otros *frameworks Java*) está basado en *custom tags*, los *tags* incluidos en las librerías *logic* o *html*, entre otras, incluyen cientos de líneas de código *Java* que ofrecen su funcionalidad a través de un simple *tag* JSP.

Los *custom tags* pueden implementarse definiendo un *TLD* y una clase *Java* que implemente la interfaz *javax.servlet.jsp.tagext.Tag*. A partir de la versión 2.0 de JSP, también se incluyen los *Tag Files* [22], que permiten crearlos de manera mucho más sencilla, utilizando un solo fichero XML con algunas directivas que vienen dadas por la especificación.

En la aplicación *Thearsmonsters*, los *custom tags* han sido implementados como *Tag Files*, y se encuentran en la carpeta */WEB-INF/tags*. Uno de ellos es

/WEB-INF/tags/window.tagx, que construye la ventana de la figura 4.24, solucionando todos los problemas anteriormente comentados.

4.5. Internacionalización

La internacionalización es el proceso de diseñar software de manera tal que pueda adaptarse a diferentes idiomas y regiones sin la necesidad de realizar cambios de ingeniería ni en el código. La localización es el proceso de adaptar el software para una región específica mediante la adición de componentes específicos de un *locale* y la traducción de los textos, por lo que también se le puede denominar *regionalización*. No obstante la traducción literal del inglés es la más extendida. [30]

Varios de los puntos que cubre la internacionalización son los siguientes:

- Varios idiomas disponibles.
- Diferentes convenciones culturales.
- Zonas horarias.
- Formato de horarios.
- Formato de fechas.
- Monedas internacionales.
- Sistema de pesos y medidas (pulgadas/centímetros, libras/gramos, etc.).
- Códigos de caracteres (*Unicode* resuelve fácilmente este problema).
- Formato de números (puntos decimales, separadores de miles, etc.).

En *Thearsmonsters* se van a utilizar los mecanismos de *Struts* y *JSTL* para internacionalizar la aplicación. En el diseño gráfico también se tiene en cuenta la internacionalización, por ejemplo, procurando no incluir textos que necesiten ser

cargados como imágenes (ya que mantener imágenes en varios idiomas es mucho más costoso que traducir un texto más).

4.5.1. Internacionalización de aplicaciones con *Struts*

En *Struts*, los mensajes que se muestran en la vista pueden ser cargados desde un fichero específico para el *locale* actual, que se guarda en la sesión y se puede cambiar desde cualquier acción del controlador.

Es necesario un fichero de recursos por cada uno de los idiomas que se van a soportar en la aplicación. Cada uno de los ficheros tiene el nombre “messages.properties” y se le añade el código de idioma al final. Por ejemplo: “messages_es.properties” para el castellano y “messages_en.properties” para el inglés. En este caso “es” y “en” son el código de los idiomas español e inglés respectivamente (según el estándar *ISO 639*). En el fichero de configuración de *Struts struts-default.xml* se debe indicar que utiliza ficheros de recursos: `<message-resources parameter='view.messages' type="props" />`. Al cambiar de idioma desde la aplicación el controlador se encargará de concatenar el código de idioma al nombre del fichero para recuperar el texto adecuado. [23]

Cuando un usuario se conecta se comprueba el idioma del navegador y se establece ese *locale* en la sesión gracias a `ThearsmonstersDefaultAction` (4.21). Si es un idioma que no está reconocido por la aplicación entonces se utilizan los mensajes por defecto (de `Messages.properties`). Si el usuario se identifica (acción `LoginAction`) también se puede fijar el *locale* a partir de los datos de ese usuario `country` y `language` (figura 4.9), dándole prioridad a esta característica sobre las preferencias del navegador.

4.5.2. Internacionalización y Localización con JSTL

`Messages.properties` sólo resuelve un aspecto particular de la internacionalización de aplicaciones: la impresión de mensajes en distintos idiomas. En *Thearsmonsters* también es necesario formatear y tratar fechas, horas, números, cantidades monetarias, etc. JSTL proporciona tags para ello (también sirven los paquetes

`java.text` y `java.util`).

Una de las librerías englobadas en JSTL es `fmt` [31], que sirve para internacionalización y formateo de las cadenas de caracteres como cifras. Para entender la importancia de su uso se muestra un pequeño ejemplo:

Mensaje dentro de un trozo de código HTML

```
<div>
    <h3>Texto 56.377.482</h3>
    <p>En un lugar de la mancha cuyo nombre no quiero ...</p>
</div>
```

Aquí se muestran los mensajes tal cual, y no se puede internacionalizar. Si se utiliza la internacionalización de *Struts*, se puede tener en el fichero *Messages_es.properties* las siguientes entradas:

`Messages_es.properties`

```
textHeader=Texto {0}
textContent=En un lugar de la mancha cuyo nombre no quiero ...
```

Y entonces el código anterior se escribiría de la siguiente forma:

Mensaje dentro de un trozo de código HTML (con JSTL)

```
<div>
    <h3>
        <fmt:message key="textHeader">
            <fmt:param><fmt:formatNumber value="56377428"/></fmt:param>
        </fmt:message>
    </h3>
    <p><fmt:message key="textContent"/></p>
</div>
```

La etiqueta `fmt:message` permite localizar un mensaje en el *locale* actual (supuestamente “es”), y opcionalmente pasarle parámetros (con `fmt:param`). Por otro lado, el número que se pasa por parámetro también se puede localizar utilizando `fmt:formatNumber`, el cual, entre otras cosas, le añade los separadores de miles (que en castellano son puntos, pero en inglés, por ejemplo, son comas). Con

la librería `fmt` también se pueden formatear fechas (`fmt:formatDate`) y otras funcionalidades relacionadas con el formateo y la internacionalización.

Aunque también se pueden localizar mensajes y números desde el código del modelo o del controlador, lo mejor es dejar los aspectos de la internacionalización como parte de la presentación, y por lo tanto, abordarlos en mayor medida en la vista.

Cuando hay objetos que deben ser autodescriptivos (que puedan mostrar mensajes por sí mismos, liberando a la vista de tener que identificarlos), la descripción que se obtiene debe estar internacionalizada, y esto se puede complicar cuando se aplica sobre composiciones de objetos, como es el caso de los atributos compuestos de un monstruo. En la figura 4.25 se muestra el método que se utiliza para que los atributos puedan ser autodescriptivos. De esta forma, cuando se añadan nuevos tipos de atributos en el juego, éstos se muestran traducidos en la vista de forma automática, solamente con añadir los textos en los ficheros `Messages.properties` para cada idioma con las claves adecuadas.

Como ilustra el esquema de la figura 4.25, los objetos de la clase `AttributeVO` implementan el método `getDescription` de modo que, en lugar de mostrar la descripción directamente, devuelve un `array` de `Strings` con los elementos que forman la descripción. En el caso de los atributos simples (`AttributeBase`), el `array` solamente contiene un identificador (suficiente para encontrar la clave con su descripción en el fichero de mensajes). Los atributos compuestos (`AttributeCompose`) devuelven la fórmula que les da valor, y para ello necesitan mostrar el nombre de los atributos simples de los que están formados. Desde la vista se sigue un procedimiento bastante simple: Si el `String` comienza por “(loc)_”, se localiza el mensaje con lo que resta de elemento (“(loc)_strength” se muestra utilizando la etiqueta `<fmt:message key='strength' />`), sino se muestra directamente el `String`. Con la descripción y un poco más de HTML se construye la vista de un atributo (utilizando más métodos de `AttributeVO` que no se muestran en este esquema, que se pueden ver en la figura 4.13). Como para cada atributo hay que hacer lo mismo, se encapsula esta parte de la vista en un `custom tag` (el fun-

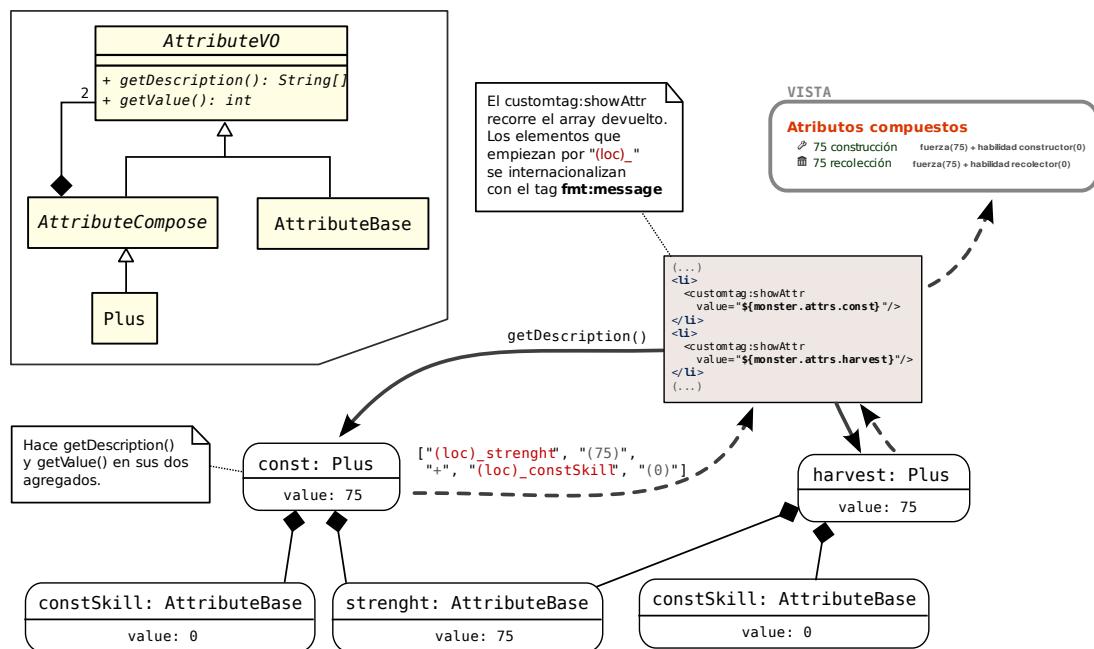


Figura 4.25: Método para mostrar las descripciones internacionalizadas de los atributos en la vista.

cionamiento y utilidad de los `custom tags` se explica en la sección 4.4.5). Si en el futuro se cambia el método de traducción de los atributos, o de su presentación, solamente habrá que modificar el `customtag:showAttr` (y en tal caso las clases *Java* que forman los atributos).

Existen muchos más ejemplos como el de la figura 4.25 en el código de *Thearsmonsters*, pero con éste basta para entender cómo se internacionaliza, en gran medida, el resto de objetos autodescriptivos del modelo.

4.6. Vistas interactivas con *JavaScript*

JavaScript es un lenguaje de programación interpretado, es decir, que no requiere compilación, utilizado principalmente en páginas Web, con una sintaxis semejante a la del lenguaje *Java* y el lenguaje *C*. [17]

Todos los navegadores modernos interpretan el código *JavaScript* integrado dentro de las páginas Web. *JScript* es la implementación de *Microsoft*, muy similar al *JavaScript* de *Netscape*, pero con ciertas diferencias en el modelo de objetos del navegador que hacen a ambas versiones con frecuencia incompatibles.

Para evitar estas incompatibilidades, el *World Wide Web Consortium* diseñó el estándar *Document Object Model* (abreviado como DOM, que significa Modelo de Objetos del Documento), que incorporan la mayoría de navegadores.

JavaScript ha tenido durante mucho tiempo la reputación de lenguaje torpe e inadecuado para el desarrollo serio. Esto ha sido en gran parte debido a las implementaciones incompatibles del lenguaje y del DOM en varios navegadores, y al uso extenso de código “pegado” lleno de errores por parte de los aficionados. Los errores en tiempo de ejecución eran tan frecuentes y difíciles de solucionar que pocos programadores intentaban corregirlos.

La reciente aparición de navegadores con un soporte adecuado de los estándares Web, *frameworks* para *JavaScript* tales como *Prototype* o *JQuery*, y herramientas de depuración de alta calidad han facilitado enormemente la creación de código organizado y escalable en *JavaScript*, y además la aparición de *AJAX* lo

ha hecho esencial. Mientras hasta hace poco *JavaScript* era solo utilizado para las tareas relativamente simples y no críticas (validación de formularios y decoraciones llamativas), actualmente se está utilizando para escribir código complejo que a menudo es responsable de buena parte de la funcionalidad básica de un sitio. Los errores en tiempo de ejecución y el comportamiento imprevisible ya no son molestias de menor importancia; son errores fatales.

4.6.1. Importancia de la librería *JQuery*

Las páginas Web que no hacen uso de *JavaScript* a menudo tienen una interfaz simple, sin animaciones, sin efectos, sin validaciones cliente y sin *Ajax*. En general, lo único que pueden hacer es enviar formularios y abrir nuevos enlaces, siempre debiendo recargar toda la página. Hasta hace poco, tratar de mejorar la interfaz directamente programando en *JavaScript* podría suponer demasiadas complicaciones (problemas de compatibilidad entre navegadores y errores muy difíciles de corregir). Actualmente existen algunas librerías con funciones y abstracciones enfocadas a poder manipular el DOM de la página, añadir efectos y dar soporte *Ajax* en la parte cliente. También se las conoce como librerías *cross browser*, dado a que una de sus principales características es que funcionan de la misma forma en los distintos navegadores para los que están diseñadas (por ejemplo *Internet Explorer*, *Firefox*, *Safari*, *Opera*, *Chrome*, etc).

JQuery [6], al igual que otras bibliotecas, ofrece una serie de funcionalidades basadas en *JavaScript* que de otra manera requerirían de mucho más código. Es decir, con las funciones propias de esta biblioteca se logran grandes resultados en menos tiempo y espacio. Sus características principales son las siguientes:

- Selección de elementos del DOM.
 - Interactividad y modificaciones del árbol DOM, incluyendo soporte para CSS 1-3 y un plugin básico de *XPath*.
 - Eventos.
-

- Manipulación de la hoja de estilos CSS.
- Efectos y animaciones.
- Simplifica enormemente el uso de *Ajax* [28] en la parte cliente.
- Soporta extensiones (hay cientos de *plugins* que aumentan las prestaciones de *JQuery*).
- Utilidades varias como obtener información del navegador, operar con *Objetos* y *Arrays*, función `trim()` (elimina los espacios en blanco del principio y final de una cadena de caracteres), etc.

Sin embargo, sigue habiendo algunos problemas típicos de *JavaScript* que no se solucionan con el uso de una librería como *JQuery*: hay que seguir teniendo cuidado con las variables globales (conviene encapsularlas dentro de un objeto único), las funciones de *JQuery* se combinan con otras funciones *JavaScript* que pueden ser problemáticas, *JQuery* no puede evitar que haya más código en el *script* que no son estándar, etc. Por eso sigue siendo recomendable minimizar el uso de *JavaScript* en la interfaz, además *JavaScript* es una tecnología “no accesible”, y aunque *JQuery* también ayude a facilitar la accesibilidad de la aplicación, conseguir que la página siga siendo perfectamente usable con *JavaScript* deshabilitado en el navegador sigue siendo bastante complicado.

4.6.2. Uso de *JavaScript* en *Thearsmonsters*

En *Thearsmonsters* se incluye algo de código *JavaScript*, sin ser exhaustivo, y siempre apoyado en las funciones de la librería *JQuery*. Concretamente, se utiliza *JavaScript* para las siguientes acciones:

- Menú desplegable en la parte superior. Cuando se pasa el ratón por encima del símbolo del juego, se despliega una lista de enlaces, y cuando se aparta, al cabo de medio segundo, se vuelve a ocultar.
-

- Cuenta atrás para aquellos eventos por los que el usuario necesita esperar. Por ejemplo, cuando se incuba un huevo de monstruo, se marca el tiempo que falta para que termine de incubarse (en el formato *hh:mm:ss*). La cuenta atrás se actualiza segundo a segundo hasta llegar a cero, momento en el que se recarga la página automáticamente.
- En algunos formularios se asiste dinámicamente con información relevante. Por ejemplo, en el formulario para el cambio de recursos (basura por dinero y viceversa) se indica la cantidad que se obtendría al realizar el cambio antes de pulsar el botón de “aceptar”, descontando automáticamente la comisión cobrada en función del nivel de las salas implicadas y de la cantidad a cambiar. También se avisa si la cantidad introducida es incorrecta (menor o igual a cero o mayor que la capacidad de almacenamiento del almacén).
- Las imágenes de los monstruos se pueden superponer al resto del *layout*, dando la sensación de que se salen por fuera de la ventana que los contiene. Esto permite una buena expresividad en el diseño gráfico, pero si no fuese por el uso de *JQuery* no sería nada recomendable intentar implementar este sencillo efecto.
- Recortar algunas descripciones largas dependiendo del tamaño de la ventana que las contiene, para asegurar que se muestren en una sola línea (y añadiendo “...” al final).
- Mensajes de ayuda no intrusivos, que aparecen cuando se pasa el ratón por encima de algún ícono, texto, ventana, o cualquier cosa que necesite una explicación. Utilizando un *plugin* de *JQuery* llamado *Tooltip*, se puede crear este efecto simplemente con añadir el atributo **title** al elemento del DOM donde se desee mostrar la ayuda. Así el contenido de la ayuda puede estar internacionalizado, y en caso de que *JavaScript* esté deshabilitado en el navegador sigue funcionando igual (aunque ya no sea con esa animación tan elegante).

- Mensajes de éxito y mensajes de error después de las acciones *Struts*. Aunque este mecanismo es mucho más profundo (también engloba al controlador), la parte que se encarga se posicionan los mensajes en la vista y de mostrarlos con un efecto suave es en *JavaScript*.

4.7. Acciones *Ajax*

En el apéndice B se haya una breve descripción de este término. También hay webgrafía útil en [28] y [13].

En la parte de la interfaz gráfica del juego dedicada a la asignación de tareas es necesario el uso de *Ajax* porque la mecánica de juego en *Thearsmonsters* se basa prácticamente en la asignación de tareas (cada monstruo tiene muchos turnos en los cuales puede cambiar de tarea).

El enfoque clásico requiere que se recargue toda la página cada vez que se cambia una tarea y esto puede ser muy lento y tedioso para el usuario. En lugar de ello se habilita una ventana (con *JavaScript*) donde se pueden ver las tareas disponibles y asignar una de ellas, de forma rápida, sin recargar el resto de la página.

En la parte servidora esto se corresponde con los casos de uso **sugerir tareas** (*SuggestTasksAjaxAction*) y **cambiar tarea**, (*SetTaskAjaxAction*) que se pueden ver en la figura 4.21, las cuales en lugar tener que devolver una página HTML completa (como el resto de acciones) solamente deben devolver el trozo de HTML necesario para poder actualizar la ventana que se muestra en el navegador.

4.7.1. Elección del método para implementar *Ajax* en *Struts*

Existen realmente muchas formas de implementar *Ajax* sobre el *framework Struts*, entre las opciones disponibles podemos encontrar *DWR* (*Direct Web Remoting*), *Echo2*, *Dojo*, *Script.aculo.us*, *Prototype*, *JQuery*, *AjaxLib*, *AjaxAnyWhere*, *ajaX JSF*, etc. La última versión de *Struts* también incluye soporte nativo

para *Ajax*, aunque para el desarrollo de este proyecto la versión de *Struts* que se utiliza es anterior.

Antes de elegir una opción hay que tener claro lo que se quiere conseguir. Algunas de estas opciones solamente sirven para interacciones *Ajax* de propósito general (como búsquedas, menús desplegables o campos autocompletados). Otras simplemente ayudan a insertar datos en el DOM de la página. Lo que interesa en este caso es una solución que permita trabajar con la interfaz (manipulación del DOM), que pueda actualizar contenidos internacionalizados y que permita reutilizar componentes de la vista¹⁸.

Sin embargo la mayoría de opciones obligan a reconstruir la vista desde *JavaScript*. Como con *DWR*, que incorpora librerías que permiten crear servicios en el servidor que devuelven objetos *Java*, los convierte de modo transparente a objetos *JavaScript* e incluye unas funciones sencillas para recorrer el DOM de la página y actualizar valores con los datos recibidos. Con esta solución generalmente hay que construir el HTML que se actualiza en la vista dos veces, una vez en el *JSP* que construye la página completa y otra vez desde *JavaScript*, además dificulta mucho la internacionalización de los contenidos actualizados.

En la sección 4.6 se ha visto que para implementar la mayoría de *scripts* necesarios en la interfaz gráfica se utiliza *JQuery*. Esta librería para *JavaScript*, además de incluir efectos y manejadores de eventos, contiene un buen soporte para *Ajax* en la parte cliente. Es decir, que sirve para realizar una llamada asíncrona remota, recibir datos en alguno de los formatos soportados y actualizar el DOM de la página de manera realmente sencilla y elegante.

Por lo tanto, utilizando *JQuery*, solamente es necesario añadir soporte *Ajax* en la parte servidora. Con *Struts* resulta relativamente simple hacer acciones que devuelvan otro tipo de contenido (por ejemplo *XML* o *JSON*, ambos formatos reconocidos por las funciones *Ajax* de *JQuery*). Pero aún devolviendo datos en *JSON* no se arregla el problema de tener que construir la vista dos veces (desde

¹⁸Siempre que sea posible, se debería aplicar el principio DRY (*Don't Repeat Yourself*. Según este principio ninguna pieza de información debería estar duplicada nunca debido a que la duplicación incrementa la dificultad en los cambios y evolución posterior.

la página *JSP* y desde *JavaScript*, aunque con *JQuery* es mucho más sencillo y limpio generar HTML para insertarlo en el DOM que, por ejemplo, con *DWR*).

Como se ha visto en la sección 4.4.1, se incluye el *plugin* de *Struts Tiles*, que permite construir una vista a base de plantillas. Utilizando este método, se podría crear una plantilla distinta para las acciones *Ajax*. Los componentes que se utilizan en las plantillas pueden ser reutilizados tanto para construir la página entera (primera petición) como para construir solamente un trozo de HTML (actualización mediante *Ajax*). De este modo ya no es necesario tener que generar el HTML otra vez desde *JavaScript*, y además así se pueden reutilizar todos los componentes JSTL que dan soporte a la internacionalización de la aplicación.

Entonces la solución propuesta para la aplicación *Thearsmonsters* es crear y utilizar un *layout* especial para crear servicios Web que devuelvan trozos de HTML en la parte servidora, y en la parte cliente se utiliza *JQuery* para realizar la petición asíncrona y actualizar un elemento del DOM con el trozo de HTML recibido¹⁹. En la figura 4.21 se puede contemplar como se han creado un tipo especial de acciones en *Struts* que generalizan las responsabilidades de las acciones de tipo *Ajax*.

4.7.2. Ejemplo de implementación con el método *Ajax* seleccionado

A continuación se muestra ejemplo a nivel de implementación para conseguir una estructura que permita realizar interacciones *Ajax* en *Struts*, que devuelva trozos de HTML internacionalizados, escritos con *JSP*, permitiendo importar archivos *.jsf*²⁰ reutilizables. Este ejemplo de codificación es relativamente sencillo y además comprende fragmentos de código *Java* (del controlador), *JSP* (de

¹⁹Este mecanismo, una vez implementado, es muy parecido al sistema de *partials* que se utiliza en el *framework Ruby on Rails*. Esta forma de actualizar los elementos de la vista resulta muy intuitiva es fácil de mantener.

²⁰En las páginas *JSP* se puede utilizar la directiva *include*, que permite incorporar el código de otros archivos como si fuese copiado-pegado. Los archivos incluidos suelen llevar la extensión “*.jsf*” (de *JSP Fragment*).

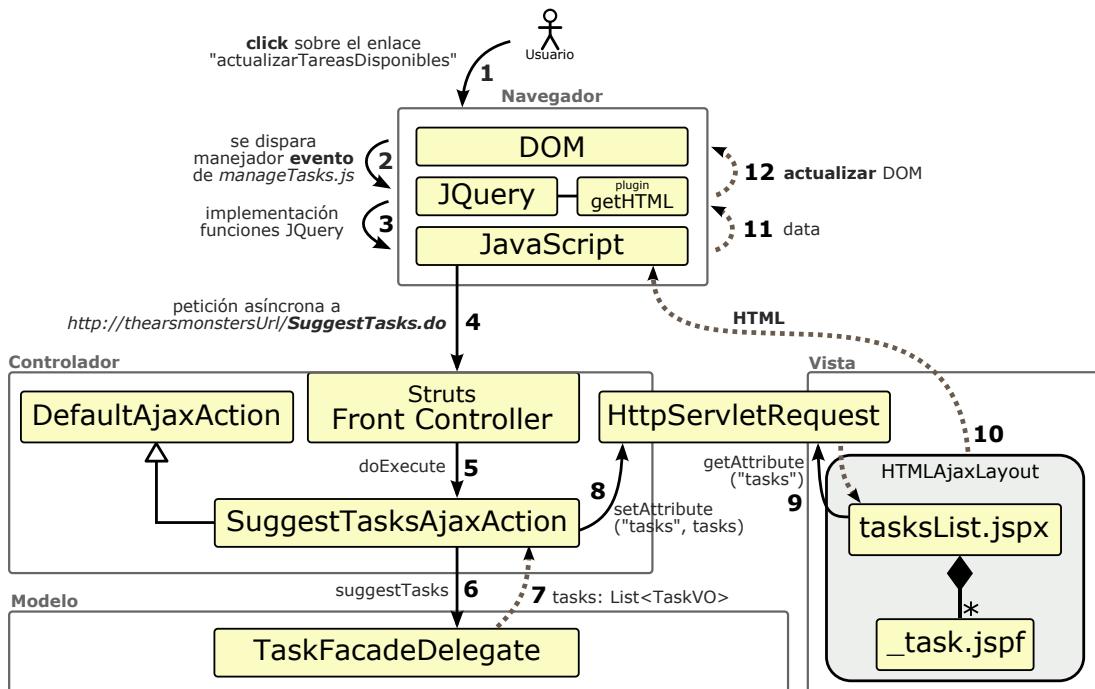


Figura 4.26: Diagrama resumen de las interacciones necesarias para implementar el caso de uso de **sugerir tareas** mediante *Ajax*.

la vista) y *JavaScript* (de la vista), que son los lenguajes más utilizados en la implementación de esta aplicación.

Nota: *Los trozos de código que se van a mostrar no son idénticos a los que hay en la aplicación real. Están escritos sobre un ejemplo simplificado con el fin de ser más legibles. Además solo se incluyen aquellos fragmentos de código que son estrictamente relevantes para cada situación.*

La acción *Ajax* que se va a implementar se corresponde con el caso de uso **sugerir tareas** (figura 4.26), donde se deben mostrar, en una ventana emergente, las tareas disponibles para un monstruo durante un turno determinado cuando se pulse sobre el elemento con `id='actualizarTareasDisponibles'`, y debe reemplazar el contenido del elemento con `id='ventanaEmergente'`, enviando los parámetros `idMonstruo` y `turno`, de los cuales depende la lista obtenida.

4.7.2.1. Código *JavaScript*

La función de *JQuery* `$.ajax(options)` tiene todo lo necesario para hacer la petición asíncrona al servidor, pero para simplificar las cosas, lo mejor es hacer un nuevo *plugin JQuery* (muy sencillo) llamado *getHTML* (figura 4.26) que añade la función `$.getHTML(options)` a la librería para que baste con indicar la `url` donde se encuentra el servicio Web y el `id` del elemento del DOM donde se quiere reemplazar el HTML obtenido²¹. El código del *plugin* creado no es necesario en este ejemplo (si se desea leer, se encuentra en el código de la aplicación en `/javascript/lib/jquery.getHTML.js`).

Por lo tanto, en la vista que contiene el DOM de la página (antes de realizar la petición *Ajax*), solamente es necesario incluir en la cabecera de la página algo como:

Dentro de la vista HTML (archivo .jspx)

```
<script type="text/javascript"
       src="javascript/jquery.getHTML.js"></script>

<script type="text/javascript"> $(function() {
    $('#actualizarTareasDisponibles').click(function() {
        $.getHTML({
            url: "http://myApp/SuggestTasks.do",
            data: {idMonstruo: 233, turno: 5},
            update: "ventanaEmergente"
        });
    }
});</script>
```

Esto añade el código *JavaScript* necesario para incluir el *plugin getHTML* y utilizarlo para hacer una petición *Ajax* a la acción *SuggestTasks.do* cuando se active el evento `click`.

²¹Lo mismo que hace este pequeño *plugin* se puede conseguir con otros *plugins jQuery* de terceros destinados a simplificar las peticiones *Ajax* (por ejemplo <http://maxblog.me/ajaxify-jquery-plugin> y <http://www.andreacfm.com/os/ajaxcontent>). Para este caso contienen demasiadas características (pretenden abarcar casos muy generales). De todas formas pueden ser una buena alternativa, y funcionan perfectamente con la parte servidora que se va a implementar.

4.7.2.2. Configuración en Struts-config y tiles-def

En la parte del servidor, se recibirá una petición a la acción `SuggestTasks.do` (que se incluye en la *url* de la petición), solamente hay que añadir las siguientes líneas de configuración a los respectivos ficheros:

struts-config.xml

```
<action path="/SuggestTasks">
    type="myAppPackages.actions.ajax.SuggestTasksAjaxAction">
        <forward name="ajaxTile" path=".tasksList" />
</action>
```

tiles-def.xml

```
<!-- Plantilla base para todas las acciones ajax -->
<definition name=".HTMLAjaxBase"
    page="/commonTiles/layouts/HTMLAjaxLayout.jspx">
    <put name="content" value="\${content}" />
</definition>

<definition name=".tasksList" extends=".HTMLAjaxBase">
    <put name="content" value="/ajax/tasksList.jspx" />
</definition>
```

En el primero se registra una acción *Struts* (al igual que con las demás acciones no *Ajax*).

En *tiles-def* se está diciendo que hay un *layout* para las vistas *Ajax* llamado `.HTMLAjaxBase`, y en concreto se pone la nueva `.tasksList` como contenido del *layout* general. Luego se verán cómo son las JSP de estas páginas.

4.7.2.3. Acciones Struts

En `struts-config.xml` se hace referencia a la acción `SuggestTasksAjaxAction` (figuras 4.26 y 4.21). Esta acción implementa a una clase abstracta común que se utilizará para todas las peticiones *Ajax*, que se llama `DefaultHTMLAjaxAction`. Por motivos de organización y limpieza, se meterán todas las acciones del tipo *Ajax* en el paquete `ajax`.

```
DefaultHTMLAjaxAction.java
public abstract class DefaultHTMLAjaxAction extends DefaultAction {

    public ActionForward doExecute(ActionMapping mapping,
                                   ActionForm form, HttpServletRequest request,
                                   HttpServletResponse response)
        throws IOException, ServletException {

        /* do Action */
        try {
            doExecuteAjax(mapping, form, request, response);
            return mapping.findForward("ajaxTile");

        } catch (InternalErrorException exception) {

            /*
             * Log error, even with debug level <= 0,
             * because it is a severe error.
             */
            ServletContext servletContext =
                servlet.getServletConfig().getServletContext();
            servletContext.log(exception.getMessage(), exception);

            // Return the standard ajax load internal error
            return mapping.findForward("AjaxHTMLInternalError");
        }
    }

    protected abstract void doExecuteAjax(ActionMapping mapping,
                                          ActionForm form, HttpServletRequest request,
                                          HttpServletResponse response)
        throws IOException, ServletException, InternalErrorException;
}
```

La acción que carga la lista actualizada que se quiere enviar a la vista, solamente tiene que hacer la llamada correspondiente al modelo (sobre la fachada

`TaskFacadeDelegate`, que se puede ver con detalle en la figura 4.16) y meter el resultado en la `request`. En esta implementación, cuando hay un error interno se lanza un `InternalErrorException`, y la acción por defecto `DefaultHTMLAjaxAction` ya se encarga de manejarlo.

```

SuggestTasksAjaxAction.java

public class SuggestTasksAjaxAction extends DefaultHTMLAjaxAction {
    protected void doExecuteAjax(ActionMapping mapping,
        ActionForm form, HttpServletRequest request,
        HttpServletResponse response)
        throws IOException, ServletException, InternalErrorException {

        // Get request parameters
        String idMonster = request.getParameter("idMonster");
        String turn = request.getParameter("turn");

        // Call model action
        TaskFacadeDelegate delegateTask =
            TaskFacadeDelegateFactory.getDelegate();
        List<TaskVO> tasks = delegateTask.suggestTasks(monsterId, turn);

        // Set request attributes
        request.setAttribute("tasks", tasks);
    }
}
```

`SuggestTasksAjaxAction` lo único que tiene que hacer es leer los parámetros de la `texttrequest` (que han sido enviados con el método `getHTML` del *plugin JQuery* desde la vista, sección 4.7.2.1), utilizarlos para hacer la llamada al modelo y meter el resultado obtenido en la `request` para que posteriormente pueda ser accedido por las páginas JSP de la vista.

4.7.2.4. Páginas JSP de la vista

Se corresponden con la *vista* del diagrama de la figura 4.26.

En `tiles-defs.xml` se indica que hay un *layout* común para las páginas que devuelven trozos de HTML mediante peticiones *Ajax* (`HTMLAjaxLayout.jspx`²²). Este *layout* se encarga de establecer el *locale* (para poder utilizar correctamente el `tag fmt:message` de JSTL²³) y de establecer el `contentType` a `text/html`. Esto último es muy importante para que el código del *script JQuery* en la vista pueda interpretar correctamente el formato de los datos recibidos.

En la aplicación *Thearsmonsters*, los *layouts* están en la carpeta `/common-tiles/layouts`, pero se pueden poner en cualquier otro sitio (realmente depende de la ruta indicada en en `tiles-defs.xml`).

```
/common-tiles/layouts/HTMLAjaxLayout.jspx
```

```
<html:html
    xmlns="http://www.w3.org/1999/xhtml"
    xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:fmt="http://java.sun.com/jsp/jstl/fmt"
    xmlns:tiles="http://struts.apache.org/tags-tiles"
    locale="true">

<jsp:output doctype-root-element="html"
    doctype-public="-//W3C//DTD_XHTML_1.0_Strict//EN"
    doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
    omit-xml-declaration="true"/>

<jsp:directive.page
    contentType="text/html; charset=iso-8859-1"/>

<fmt:setLocale
    value="\${sessionScope['org.apache.struts.action.LOCALE']}"
    scope="session"/>
```

²²Las páginas `.jspx`, son como las clásicas `.jsp` pero escritas en notación XML estándar. Realmente hay una equivalencia (que es fácil de suponer) entre unas y otras, por lo que este código puede ser fácilmente traducido al JSP habitual.

²³En la sección 4.4.5 se hace una breve introducción a JSTL.

```
<tiles:get name="content"/>
```

```
</html:html>
```

Y por último, el contenido específico de la lista que se quiere mostrar, dentro de la carpeta */ajax* (también se puede poner en cualquier otro sitio).

```
/ajax/tasksList.jspx
```

```
<jsp:root xmlns="http://www.w3.org/1999/xhtml"
           xmlns:jsp="http://java.sun.com/JSP/Page"
           xmlns:c="http://java.sun.com/jsp/jstl/core"
           version="2.0">

    <ul class="tasksList">
        <c:forEach items="${tasks}" var="task" ><li>
            <jsp:directive.include file="/../_task.jspf"/>
        </li></c:forEach>
    </ul>

</jsp:root>
```

En este último fragmento de código JSP, se muestra únicamente una lista (de clase ‘‘*tasksList*’’) en la cual se muestran las tareas que hay como atributo *tasks* de la *request*. Para mostrar cada elemento de la lista se incluye el archivo */../_task.jspf* (cuyo código no es necesario mostrar) que se encarga de formatear la vista de la tarea que esté disponible como atributo *task* (además lo hace en el idioma actual). El archivo */../_task.jspf* se podrá reutilizar desde cualquier vista que necesite mostrar el contenido de una tarea, del mismo modo que esta lista de tareas (*/ajax/tasksList.jspx*) también se podrá insertar como contenido en cualquier *layout* que lo necesite (da igual que sea una petición *Ajax* o una petición normal de *Struts*).

4.7.2.5. Añadir más acciones *Ajax* sobre esta estructura

A partir de aquí, añadir nuevas acciones *Ajax* es mucho más sencillo, pues ya están definidos tanto la acción *Ajax* de *Struts* por defecto como el *layout*.

De el diagrama de la figura 4.26 solo habría que cambiar el manejador del evento, la clase `SuggestTasksAjaxAction` (con el código del controlador para el nuevo caso de uso, y si es necesario habrá que añadir más métodos en las fachadas del modelo) y el contenido del `HTMLAjaxLayout`.

Para crear una nueva acción *Ajax*, hay que declarar las acciones en el fichero de *Struts struts-config.xml*, el contenido de las páginas en `tiles-defs.xml`, añadir una acción *Struts* que extienda a `DefaultHTMLAjaxAction` (hasta aquí es lo mismo que necesita cualquier acción que no sea *Ajax*) y desde *JavaScript* utilizar la función `$.getHTML` (implementada en el *plugin* de *JQuery getHTML*) para realizar la petición *Ajax* y poner el contenido obtenido dentro del elemento del DOM que se quiera actualizar.

Si se desea poner un contenido HTML desde el principio (cuando se crea la página) y luego poder actualizarlo vía *Ajax*, solamente es necesario definirlo una vez (se cumple el principio *DRY*) y además no es necesario generar HTML desde *JavaScript*, sino que se hace mediante JSP. Esto se puede conseguir de dos maneras distintas:

1. Crear una página `.jspx` e importarla en la página JSP completa (utilizando la etiqueta `jsp:directive.include`). Después la página JSP *Ajax* también importa el mismo `.jspx`, pero en la *request* están los datos actualizados. Así se puede reemplazar el mismo HTML pero con los nuevos datos obtenidos del modelo.
2. Al cargar la página completa, utilizar directamente la función `$.getHTML` dentro del manejador `$.ready` de *JQuery*, de forma que justo después de cargar la página se introduce el código HTML donde sea necesario. Y luego se podrá actualizar al recibir los eventos oportunos.

La segunda opción produce un poco más de carga en el servidor (al realizar

la primera petición se necesitan hacer también las peticiones *Ajax*), pero resulta bastante más fácil de programar porque requiere menos ficheros y menos líneas de código. Aun así, en *Thearsmonsters* se ha utilizado la primera opción.

Capítulo 5

Interfaz Gráfica

Índice general

5.1.	Breve introducción al diseño Web	144
5.2.	Imagen digital	147
5.3.	Arte gráfica en <i>Thearsmonsters</i>	152
5.3.1.	Portada del juego	153
5.3.2.	Dentro del juego	155
5.3.3.	Representación gráfica de la guarida	160
5.3.4.	Representación gráfica de los monstruos	164

ESTE capítulo se centra en el diseño gráfico de la interfaz de usuario. Un buen diseño gráfico no implica unos efectos especiales de última generación, de hecho, los artistas implicados en el desarrollo del juego están siempre limitados por el nivel de detalle que pueda soportar el motor gráfico, y un buen artista debe saber como sacar el máximo rendimiento del mismo, consiguiendo crear la ambientación y el estilo característico del juego a pesar de las limitaciones.

Las ideas expuestas en este capítulo relacionadas con el diseño Web han sido obtenidas principalmente de los siguientes artículos y *blogs* en Internet: [26], [18], [16], [29], [9], [8] y [12]. Las ideas relacionadas con el diseño de videojuegos se

obtienen de [24], [25] y [10].

5.1. Breve introducción al diseño Web

Thearsmonsters es un juego basado en navegador, por lo tanto se aplican todos los aspectos relacionados con el diseño Web.

El diseño Web es una actividad que consiste en la planificación, diseño e implementación de sitios Web y páginas Web. No es simplemente una aplicación del diseño convencional, ya que requiere tener en cuenta cuestiones tales como navegabilidad, interactividad, usabilidad, arquitectura de la información y la interacción de medios como el audio, texto, imagen y vídeo.

Para hacer un buen diseño web, además de ofrecer un aspecto agradable y profesional, hay que tener en cuenta que lo importante es transmitir una idea y un sentimiento al usuario, y que todo esto se produzca de manera clara, concisa y transparente. Por lo tanto, la primera tarea del diseñador (que muchas veces se olvida) es definir cuáles son esas ideas y sensaciones que se quieren transmitir. En este proyecto por ejemplo, se intenta dar al usuario la sensación de estar en mundo subterráneo donde hay monstruos, y procurar que no sea algo oscuro y cavernoso, sino algo estético y agradable, como si de un local de moda se tratase.

En el juego *Thearsmonsters* no hay suculentas animaciones 3D, no hay libertad de diseño, ni siquiera se pueden aplicar todos los colores de una paleta convencional, y a pesar de ello el juego necesita seguir siendo expresivo.

El diseño web es un campo muy amplio que se escapa a los propósitos de esta memoria. No obstante sí que se puede hacer una muy breve introducción a los aspectos más relevantes:

- **Tipografía:** Tanto en la tipografía tradicional como en la de internet la regla principal es que todo debe ser legible. Generalmente las letras de los títulos son más llamativas y pueden permitirse un mayor nivel de detalle. Para los cuerpos de texto hay que utilizar tipografías “lisas” o más comúnmente denominadas *SansSerif*, las fuentes *Serif* (del estilo *Times*) tienen

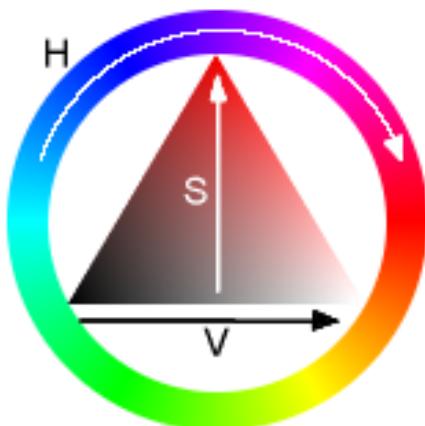


Figura 5.1: Espacio *HSV* del color.

muchos más detalles y a tamaños pequeños se hacen poco legibles en una pantalla¹.

- **Teoría del color:** Es un grupo de reglas básicas en la mezcla de colores para conseguir el efecto deseado combinando colores de luz o pigmento. Comprendiendo los diferentes modelos del color (RGB, RYB, CYM, ...), las armonías de color (aquellos que funcionan bien juntos, complementarios, tríadas, ...), los espacios de colores (HSV como en la figura 5.1, YIQ, ...) y la percepción del color por parte del ojo humano, se pueden formalizar las paletas utilizadas en un diseño. Existen algunas normas básicas aplicadas al diseño Web, por ejemplo, los colores en este medio interpretados de forma distinta por cada navegador, sistema operativo e incluso monitor, quedando a disposición del diseñador una paleta denominada *WebSafe* (desgraciadamente con solo 256 colores), creada para asegurar que los colores utilizados mantienen su armonía allá donde se visualicen.
- **Espaciado y composición:** La posición de los elementos en la pantalla es

¹Estas normas se aplican a los gráficos que son visualizados en una pantalla. La impresión en papel cuenta con un mayor nivel de detalle por lo tanto ahí sí que se pueden escribir textos largos con letras de tipo *Serif*.

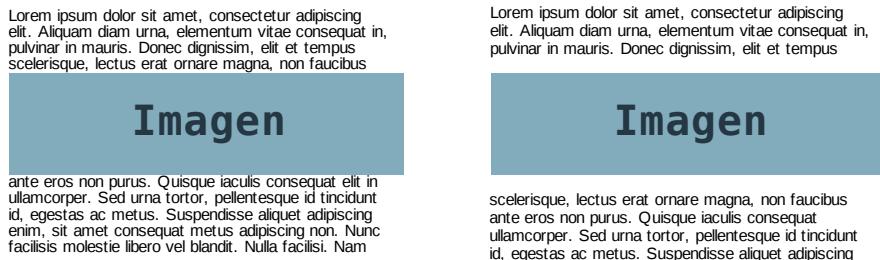


Figura 5.2: Comparación entre un párrafo sin espacios (izquierda) y otro que separa adecuadamente el texto de la imagen.

uno de los puntos más importantes a la hora de hacer un diseño, pues no es lo mismo colocar una imagen en la parte superior de la composición o en la parte inferior, con un texto a su lado o sin él, que sea una imagen grande o pequeña y así con infinidad de detalles que dotarán la presentación de su debida personalidad. Un ejemplo de buena práctica para aplicar en este ámbito puede ser la frase “*deja que tu texto respire*”, que se aplica para que los textos sean legibles. En la figura 5.2 se puede apreciar como el texto de la izquierda está muy pegado a la imagen y su interlineado es muy bajo para el tamaño de letra escogido, haciéndolo confuso para la lectura. El otro en cambio se lee perfectamente.

En la figura 5.3 se puede ver un buen ejemplo (extraído de la galería [11]) donde se compone una Web con imágenes y texto. En ella se usa eficientemente el espacio entre líneas, posicionando las imágenes por tamaño y relevancia. Los elementos más importantes tienen un color más llamativo, los colores de fondo son suaves y no entopecen la lectura del texto, se ve perfectamente donde está el menú de navegación y los detalles se enumeran en la parte inferior, permitiendo al usuario centrarse solo en aquellos aspectos que realmente le interesan. A primera vista se nota que la página tiene su propia personalidad, los colores son armónicos y la composición es limpia.



Figura 5.3: Ejemplo de buen diseño Web: Página principal de *Firefox*.

5.2. Imagen digital

Una imagen digital, también llamada gráfico digital, es una representación bidimensional de una imagen utilizando *bits* (unos y ceros). Dependiendo de si la resolución de la imagen es estática o dinámica el gráfico puede ser:

- **Gráfico rasterizado:** También llamado *bitmap*, es una estructura o fichero de datos que representa una rejilla rectangular de *píxeles* (puntos de color) que se puede visualizar en un monitor de ordenador, papel u otro dispositivo de representación. Este formato está ampliamente extendido y es el que se suele emplear para tomar fotografías digitales y realizar capturas de vídeo. Para su obtención se usan dispositivos de conversión analógica-digital, tales como escáneres y cámaras digitales.
- **Gráfico vectorial:** Es una imagen digital formada por objetos geométricos independientes (segmentos, polígonos, arcos, etc.), cada uno de ellos definido por distintos atributos matemáticos de forma, de posición, de col-



Figura 5.4: Comparación de una imagen vectorial con una rasterizada.

or, etc. Por ejemplo un *círculo rojo* quedaría definido por la posición de su centro, su radio, el grosor de línea y su color. El interés principal de los gráficos vectoriales es poder ampliar el tamaño de una imagen a voluntad sin sufrir el efecto de escalado que sufren los gráficos rasterizados. Asimismo, permiten mover, estirar y retorcer imágenes de manera relativamente sencilla. Su uso también está muy extendido en la generación de imágenes en tres dimensiones tanto dinámicas como estáticas.

Los gráficos vectoriales en general no son aptos para codificar fotografías o vídeos tomados en el “mundo real” (prácticamente todas las cámaras digitales almacenan las imágenes en formato rasterizado). Los datos que describen el gráfico vectorial deben ser procesados, es decir, el computador debe ser suficientemente potente para realizar los cálculos necesarios para formar la imagen final. Si el volumen de datos es elevado se puede volver lenta la representación de la imagen en pantalla, incluso trabajando con imágenes pequeñas. Por otro lado, las imágenes vectoriales pueden requerir menor espacio en disco que un *bitmap*. Las imágenes formadas por colores planos o degradados sencillos son más factibles de ser vectorizadas. A menor información para crear la imagen, menor será el tamaño del archivo.

Una imagen digital, ya sea vectorial o rasterizada, debe tener un formato que el dispositivo de salida sepa interpretar. Actualmente existen muchos tipos de formatos y prácticamente cada programa informático utiliza el suyo propio. Por suerte hay algunos formatos que se han hecho muy comunes y que casi todos los programas gráficos reconocen. Es habitual que un programa de diseño reconozca varios formatos de imagen y que los convierta al suyo propio para realizar la

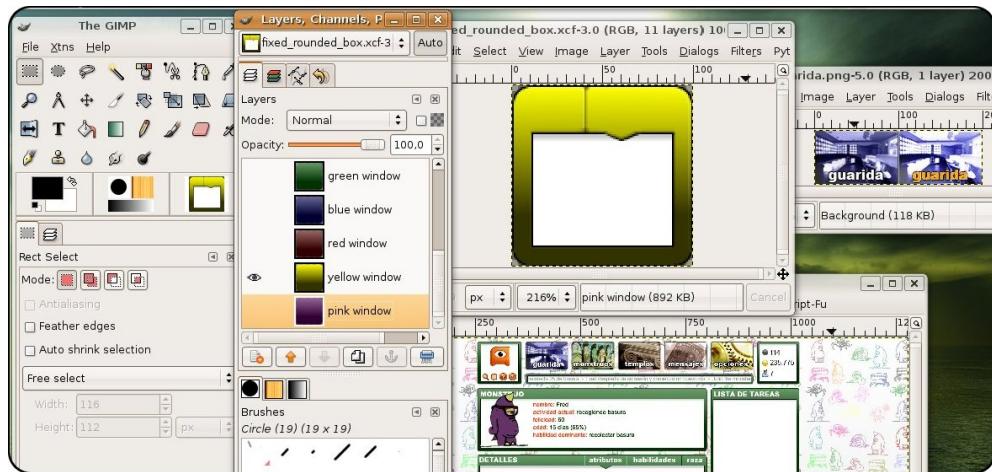


Figura 5.5: Editando gráficos con el GIMP [1]. Prácticamente todos los elementos (*backgrounds* y *sprites*) de la interfaz web se han hecho con este programa.

edición de los gráficos. Después de realizar la edición también es normal que se puedan guardar en varios formatos.

Los formatos más utilizados en la Web (reconocidos por la mayoría de navegadores) y en aplicaciones multimedia son:

- **JPG:** Formato rasterizado. Buena compresión de imagen (ocupa poco espacio). Útil para guardar imágenes reales como fotografías o gráficos complejos. Lo malo es que no permite transparencias (las imágenes son siempre opacas y rectangulares).
- **GIF:** Rasterizado. Permite transparencias pero solo sobre un color de fondo. Los *antialiasados* suelen ser bastante malos cuando se cambia el color de fondo. Permite guardar en el mismo archivo varios fotogramas, por lo que se utiliza muy amenudo para mostrar animaciones en la web. Sólo permite utilizar 256 colores.
- **PNG:** También rasterizado. Mejor compresión que el formato GIF, mucho mejor soporte para transparencias y mucha mejor calidad de color. El problema es que el navegador *Internet Explorer 6* (el más utilizado hasta

hace poco) no soportaba este formato de imagen (en realidad ahora existen métodos poco ortodoxos para conseguir que lo soporte), y por ello se utilizaba poco. Este navegador tampoco tenía soporte para gráficos vectoriales (solamente archivos *Flash*) por lo que generalmente no se utilizan imágenes en formato vectorial en la web.

También son muy populares los formatos TIF y EPS para utilizar en las impresoras. Otros formatos como PSD (Photoshop), XCF (Gimp), SVG (Inkscape) o SWF (Flash) son conocidos gracias a la popularidad de los programas que los utilizan.

Las imágenes del juego *Thearsmonsters* están en formato *JPG* y *PNG* porque se deben poder visualizar en un navegador Web². No se utiliza el formato *GIF* debido a su escasa calidad con el *antialiasado* y las transparencias.

También se guardan los archivos originales en los formatos específicos de cada editor. Como se han utilizado los programas *Gimp* [1], *Inkscape* [2] y *Photoshop* [3] hay imágenes en los formatos *XCF*, *SVG* y *PSD*, respectivamente. Es necesario guardar también las imágenes con estos formatos para facilitar el mantenimiento y los futuros cambios en las mismas ya que ahí se guarda información de capas, del historial de cambios y muchas más cosas.

Por ejemplo, si de pronto surge la necesidad de representar un monstruo al doble de tamaño y se amplía la imagen en formato *PNG* que se utiliza para mostrar en la Web se estaría perdiendo mucha calidad³. Sin embargo la versión del *Inkscape* en formato *SVG* puede ampliarse de manera ilimitada ya que es un gráfico vectorial, sin perder nada de calidad. La manera de proceder en este caso sería ampliar la imagen *SVG* con el *Inkscape* y rasterizarla de nuevo a formato *PNG* para que se pueda visualizar en la Web, sin pérdida alguna de calidad.

²Las imágenes en formato PNG no pueden visualizarse a priori en la versión 6 o anteriores de *Internet Explorer*, aunque este navegador es cada vez menos utilizado y además este problema puede solventarse con programación *JavaScript*.

³Recuerde que una imagen en formato PNG es una imagen rasterizada, y al ampliarla se van apreciando cada vez más los píxeles).

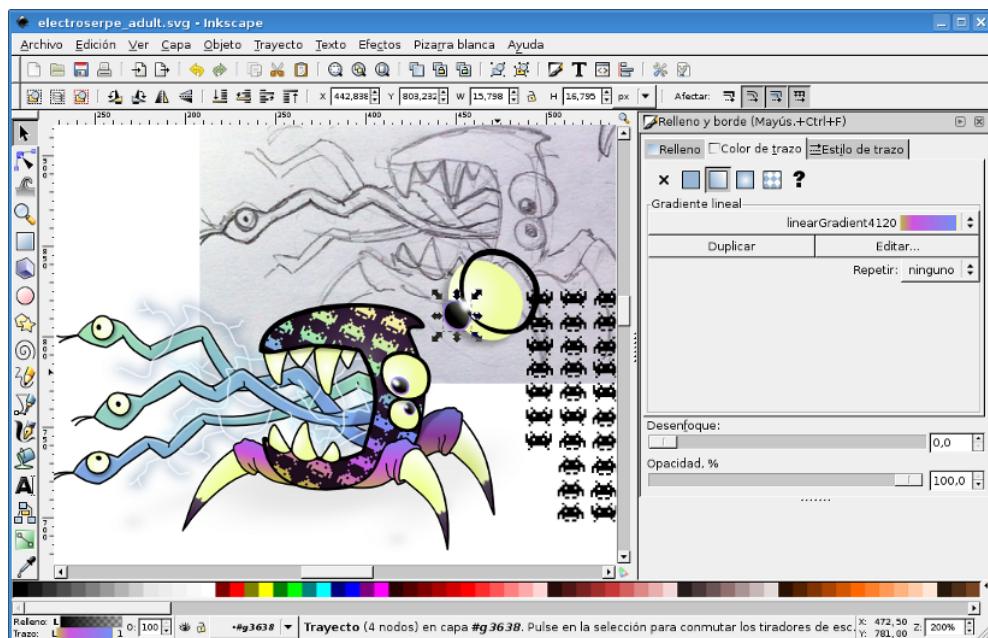


Figura 5.6: Creación del gráfico vectorial de un monstruo con el *Inkscape* [2] a partir de su diseño en lápiz escaneado.

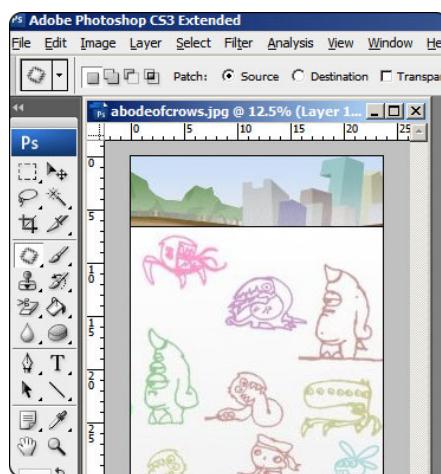


Figura 5.7: Editando la imagen de fondo del juego con *Photoshop* [3].



Figura 5.8: A la izquierda una captura del *Gears of Wars 2*. A la derecha del *Patapón*. Si en un juego no se puede crear un entorno gráfico realista, al menos debe contar con un buen diseño gráfico que sea estético, original y atractivo.

5.3. Arte gráfica en *Thearsmonsters*

En la historia de los videojuegos, generalmente existe la tendencia de que los gráficos intenten simular la realidad, cuanto más realistas mejor, ya que impresionan a los usuarios, dan categoría al *hardware* (consola) donde son ejecutados y consiguen que la experiencia de juego sea más envolvente (cuanto más realismo, más sensación de estar dentro del juego). Sin embargo esto no siempre se cumple, pues existen multitud de títulos que a pesar de su increíble apartado gráfico no consiguen transmitir ninguna sensación o experiencia de juego, ya que al final lo importante es el entretenimiento.

La decisión de intentar crear gráficos realistas depende del tipo de juego, de la plataforma y de las posibilidades de la compañía que lo desarrolle. Si se quiere crear un *shooter* de acción que trate de competir con juegos como *Halo 3* o *Killzone 2*, si no hay buenos gráficos, el fracaso está asegurado. Es más, si los gráficos de un juego de este tipo no son de última generación es probable que el proyecto no sea viable. Por otro lado, las compañías que consiguen crear un juego con un motor gráfico espectacular, seguramente puedan rentabilizar el proyecto con la venta del motor gráfico a terceros, por esto y por otras razones, las *superproducciones* de videojuegos son muy rentables.

Si no se dispone de los medios necesarios para crear unos gráficos de última generación, las alternativas se encuentran en otras plataformas, como pueden ser la *Wii*⁴, el *iPhone* o la misma *World Wide Web* (como es el caso del nuestro juego), las oportunidades de negocio residen en la originalidad y jugabilidad del producto, y para llamar la atención del usuario, a falta de un realismo asombroso, lo importante es el diseño artístico.

En *Thearsmonsters* se ha cuidado el diseño artístico (dentro de las limitaciones), ya que es la única posibilidad de situar al juego en un lugar competente dentro de la industria del entretenimiento.

A continuación se van a exponer y justificar cada uno de los apartados importantes del juego con respecto a su diseño gráfico.

5.3.1. Portada del juego

El objetivo de la portada es captar nuevos jugadores y despertar el interés por el juego ofreciendo información sobre el mismo, a la vez que debe ofrecer una vía de acceso para los jugadores que ya están registrados.

El visitante debe poder identificar qué es lo que proporciona el sitio Web, es decir, debe darse cuenta inmediatamente de que está en la portada de un juego online gratuito. Para ello se incluye de forma clara la frase “*juega ahora gratis*” dentro de un enlace hacia el registro de usuario.

La portada del juego transmite un mensaje inicial muy importante. De forma concisa se intenta decir al visitante que está apunto de comenzar una experiencia de juego distinta. Como se sabe que la principal diferencia temática de este juego con respecto a los demás es la presencia de monstruos pintorescos, se incluye un gran dibujo con varios de los monstruos del juego, junto al símbolo y logotipo. De esta manera, enseguida pueden asociar el nombre “*Thearsmonsters*” con los monstruos de la portada. También se incluye una pequeña frase diciendo que se trata

⁴Como curiosidad, comentar que gracias a las limitaciones gráficas de la *Wii* en favor de su jugabilidad intuitiva, se ha favorecido la aparición de buenos juegos de autor, muy originales y centrados en el diseño más que en el realismo, capaces competir en el mercado con los más grandes de la industria.

de un juego para navegador, sin necesidad de instalación, con cientos de usuarios en línea, etc. No debe haber demasiado texto (exceso de información), porque en ese caso lo único que se consigue es enmascarar la información importante, ésa que es capaz de captar la atención del jugador.

Una vez que el visitante ya tiene identificada la página (primer vistazo), quizás necesite ampliar su conocimiento sobre el juego antes de registrarse. Para ello se ponen en la ventana inferior una serie de pestañas que muestran diferentes contenidos relacionados con el juego. En la propia ventana se muestran más detalles sobre el mismo, aunque sean solo los aspectos principales, mostrando mensajes acompañados de pequeñas imágenes del estilo “*cría tus propios monstruos y hazlos crecer*”, “*programa sus tareas diarias y despreocúpate*”, etc.

Las demás pestañas incluyen: “capturas de pantalla” de diferentes secciones del juego, acceso directo al registro de usuarios, enlace al foro, reglas del juego, “concept art” con fondos de pantalla e imágenes artísticas, información sobre los creadores del juego, etc.

En la parte superior se incluye el formulario de identificación para aquellos usuarios que ya están registrados. Este formulario está siempre visible y, a pesar de estar siempre presente, es lo suficientemente pequeño para no llamar demasiado la atención a los visitantes que entran por primera vez (ya que lo importante en su caso que accedan al formulario de registro).

El símbolo del juego (figura 1.1) se incluye en la portada y en muchos otros lugares para que los usuarios puedan identificar claramente las referencias al juego. Su diseño es simple, haciendo referencia al estilo y filosofía que tiene el resto del juego. Su forma surge de simplificar la “A” de *ARS*, palabra comprendida dentro del título del juego *Thearsmonsters* que significa *arte* en latín⁵, donde el agujero que queda dentro de la “A” se convierte en un ojo redondo, haciendo alusión a los propios monstruos del juego.

⁵ARS también son las siglas de Ángel Romo Sandoval, coautor del diseño de los monstruos que aparecen en el juego.

5.3.2. Dentro del juego

Una vez dentro del juego, los objetivos del diseño de los contenidos son muy diferentes a los de la portada. Aquí lo que interesa es que el jugador sepa donde se encuentra, en qué sección está, que tenga información siempre disponible sobre lo que está haciendo, que pueda saber qué está sucediendo y que pueda navegar por los elementos de la interfaz del juego de forma sencilla.

Las ideas principales que el diseño de la interfaz debe expresar son:

- Estilo moderno, siguiendo las tendencias del diseño actual [20] (bordes redondeados, colores suaves, uso de iconos acompañando el texto, degradados poco intensos, etc).
 - Debe haber colores, las interfaces monocromáticas son demasiado serias para un juego de monstruos graciosos.
 - El jugador debe tener alguna noción de dónde se encuentra su guarida, por eso se intenta situar la interfaz bajo tierra, dibujando una ciudad en la parte superior de la pantalla.
 - El jugador debe saber en qué sección se encuentra (guardia, monstruos, mensajes, etc). Un buen método es otorgar un color diferente a cada sección, así cuando se encuentre en la guarida, las ventanas se ven de color azul, mientras que cuando se ven los monstruos, las ventanas son verdes. Esto también ayuda a mantener el colorido del juego.
 - Los botones del menú principal pueden ser grandes mientras sean expresivos y decorativos. Serán visibles todo el tiempo, así que es mejor que ayuden en la composición visual de la página.
 - Aparte de los botones principales, los enlaces y elementos secundarios deben aparecer sólo en el contexto donde sean necesarios, por ejemplo, a modo de pestañas en las ventanas de cada sección.
-

5.3. Arte gráfica en *Thearsmonsters*



Figura 5.9: Algunos prototipos que ayudaron a definir el *layout* del juego. El estilo y la composición logradas se han ido refinando en cada uno, tratando de enfatizar y las ideas principales que debe expresar.

- La aparición de elementos en la pantalla va de más generales a más específicos. Por ejemplo, los botones del menú principal están arriba, la guarida en el medio y al seleccionar una sala de la guarida se muestra debajo.

En la figura 5.9 se muestran algunos diseños previos que se fueron realizando para definir la interfaz. Cada uno de ellos se ha desecharido por una razón:

- **Prototipo 1:** Se trata simplemente de una prueba de estilo, maquetación, texturas, etc. Sobre él se puede establecer qué cosas pueden ser interesantes para el diseño de la web.
- **Prototipo 2:** Mantiene la esencia de que la guarida se encuentra debajo de una gran ciudad, pero el contraste es demasiado fuerte y el estilo es muy “agresivo”. Este diseño es más propio de un *portafolio* o de una página de información que de la interfaz de un juego para navegador.
- **Prototipo 3:** La estructura básica de la interfaz ya está siendo bien definida. Resulta claro y ordenado, sin embargo tiene muy poco colorido, lo cual se hace demasiado serio.
- **Prototipo 4:** En base al prototipo 3, se intenta introducir el concepto de la guarida bajo tierra y añadir más colores. Se definen las secciones por color, se hacen pruebas para ver cómo quedarían las pestañas de las ventanas y de paso se hace una prueba para la vista de guarida.
- **Prototipo 5:** Con los gradientes en los marcos de las ventanas gana profundidad y relieve. Los botones de colores tienen imágenes que los hacen menos aburridos y la imagen de la ciudad en la parte superior apenas ocupa espacio. En los prototipos anteriores habría que hacer *scroll* en casi todas las pantallas, ya que la ciudad puede llegar a ocupar casi la mitad de la ventana del navegador. Este diseño es casi definitivo, ya que sigue presentando inconvenientes: el marco de las ventanas es demasiado ancho y queda un poco “bruto”, los botones aunque son bonitos están demasiado cargados

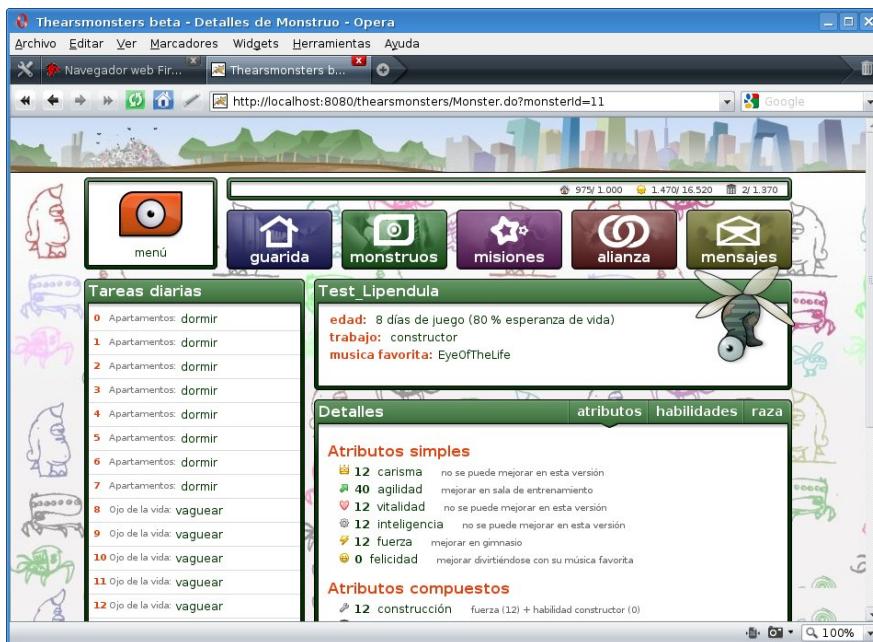


Figura 5.10: Captura de pantalla del juego durante una fase bastante avanzada del desarrollo. Ya puede apreciarse cómo va a quedar el *layout* final, y se puede comparar con los prototipos previos de la figura 5.9.

(sería mejor si incluyesen algún símbolo en lugar de tanto gráfico), el fondo de tierra tiene un color que no es armónico con el resto de la página y el menú desplegable que se supone que sale al pasar el ratón por encima del símbolo del juego es demasiado estrecho.

A partir del *prototipo 5* ya se crea una imagen bastante satisfactoria de lo que podría ser el *layout* final del juego. En la figura 5.10 se puede ver un ejemplo de cómo termina la evolución del diseño. Para llegar a este diseño final se han tomado, entre otras, las siguientes decisiones:

- El fondo no es necesario que sea de tierra (como en el prototipo 4), ya que al tener la ciudad situada en la parte superior, ya se sobreentiende que la guarida está debajo, así se obtiene un resultado mucho más armónico y luminoso.

- El diseño de los botones del menú superior tiene en cuenta el hecho de que siempre van a estar visibles, por lo que deben tener un buen nivel de detalle, no ser aburridos y concordar con el estilo del resto de la página. De hecho casi definen el estilo de la página. Es importante que no cansen con el tiempo, no deben ser excesivamente llamativos o enrevesados. Por lo tanto se ha decidido que sean grandes y cuadrados con bordes redondeados, pero mucho más sencillos y esquemáticos que en el prototipo 4. Aplicando un poco de brillo y sombra al borde de los botones se causa la sensación de que tienen volumen, así está mucho más claro que son botones.
- Diferentes colores para cada botón del menú principal. Cuando se pincha en uno de ellos, la página resultante contendrá las ventanas del mismo color. Por ejemplo, como el botón de monstruos es verde y el de mensajes es rojo, las ventanas que contengan información sobre los monstruos serán verdes y las de los mensajes rojas. Al ser los bordes de la ventana oscuros en todos los casos no se pierde el contraste de la composición, sin embargo de esta forma se orienta al usuario por la interfaz y además ofrece un resultado más colorido y alegre.
- Cada sección de la interfaz tiene su propio color (guardia azul, monstruos verde, misiones morado, alianza rojo y mensajes amarillo), aunque se mantiene el blanco como color común de fondo y el naranja como color principal y neutro. El logotipo del juego y los títulos de los párrafos son naranjas. La selección del naranja se debe a que tiene un carácter acogedor, cálido, estimulante y una cualidad dinámica muy positiva y energética. Representa la alegría, la juventud, el calor o el verano (aunque también puede expresar inestabilidad, disimulo e hipocresía). La paleta de variaciones del naranja es la misma que la del color marrón, ya que ambos colores son de naturaleza similares. Es un color que destaca mucho sobre el entorno que le rodea, por lo que se puede usar para dar un mayor peso visual a ciertos elementos de una composición, aunque hay que ser comedido en su uso, ya que si es brillante llena mucho la vista del usuario. Admite buenos degradados.

dos, y combina muy bien con su complementario (el azul), dando buenos contrastes, sobre todo cuando uno es claro y el otro oscuro.

- Los iconos orientativos que se incluyen con los textos no son ni muy grandes ni muy complejos. Están compuestos por colores poco saturados y poseen unas proporciones similares. Un ícono no puede ser más llamativo que el resto, ya que daría la sensación de que algún atributo es más importante que otro y no es así, simplemente cada uno sirve para una cosa distinta.
- Se utiliza una paleta de colores armónica para la composición final, teniendo en cuenta la saturación, el brillo y el contraste que producen los colores complementarios de los matices principales de cada sección.
- Fondo de la web claro y ventanas con bordes oscuros. En general, el texto sobre un fondo blanco es más legible y por lo tanto el lector debe forzar menos la vista. Esto también se puede observar en colores claros y oscuros, no es algo que suceda únicamente con el blanco y el negro. En este caso el fondo de las ventanas es blanco, ya que permite hacer las letras y los íconos en cualquier color con tal de que sean algo oscuros. Para dar sensación de volumen en el borde superior de la ventana, se aplica un gradiente de claro a oscuro dentro del color correspondiente a esa sección, y al fondo de la ventana se aplica una pequeña sombra. Todo esto sin incluir colores, formas o texturas que puedan ensuciar la imagen global. Se ha probado con otras alternativas pero se han desecharo porque lo importante es que llamen la atención los botones y enlaces que son significativos y no los elementos del decorado.

5.3.3. Representación gráfica de la guarida

Para representar el estado de la guarida bastaría con una lista de los nombres de las salas con sus datos (nivel, tamaño ...), sin embargo esto resulta poco gratificante, al jugador puede darle la sensación de estar ampliando una simple



Figura 5.11: Vista de guarida del juego *Dig'n'fight*.

lista de elementos, cuando lo que debería sentir es que en algún lugar del subsuelo de ese mundo virtual se está construyendo una verdadera guarida llena de vida.

La mejor idea es proporcionar una vista gráfica de la guarida⁶. Así resulta fácil imaginarse cómo sería la vida de los monstruos dentro del mundo *Thearsmonsters*, dando más carácter y estilo al juego. Los dibujos de las salas son también una especie de “premio” cuando los jugadores construyen una nueva, ya que así pueden ver de forma tangible los resultados de sus acciones.

El reto para el diseño en este caso es tratar de representar una guarida subterránea pero moderna, integrarla en el resto del diseño de la página y que sea posible de implementar dadas las limitaciones del HTML, del CSS y de los recursos técnicos disponibles.

La primera inspiración se ha encontrado en un juego que a priori parecía bastante similar a *Thearsmonsters* en cuanto a temática llamado *Dig'n'fight* (<http://www.dignfight.com>), cuya guarida se representa gráficamente como se muestra en la figura 5.11, donde el jugador maneja un puñado de monstruos que viven en una guarida bajo tierra tratando de robar recursos a las guaridas de los jugadores con puntuación similar. Sin embargo pronto se descubrió que

⁶En la sección 2.6.2 del capítulo de contextualización, se obtiene la idea de la representación gráfica para navegador del juego *Travian*, donde los jugadores pueden ver su aldea.



Figura 5.12: Boceto de la vista de la guarida en el juego.

la estética del juego es muy diferente (y no sólo eso, sino que el propio tampoco tiene nada que ver con *Thearsmonsters*), en este caso las guaridas son como minas (nada modernas), inspiradas a su vez en la estética del clásico *Dungeon Keeper* (http://es.wikipedia.org/wiki/Dungeon_Keeper). Además *Dig'n'fight* no hay noción geográfica (no puedes ver donde están tus vecinos).

En *Thearsmonsters* las guarida se suponen que están divididas en bloques de pisos (en cada piso hay una guarida). Por lo tanto parece razonable representar la guarida desde una vista lateral, como un pasillo con puertas, donde cada puerta es la entrada a una sala. Una de esas puertas son unas escaleras o un ascensor que lleva a las guaridas vecinas (así la navegabilidad entre guaridas resulta más intuitiva). La representación lateral también tiene la ventaja de que a medida que se construyen nuevas salas se van incorporando a la derecha, dando la sensación de que la guarida se está ampliando.

La imagen de la figura 5.12 es una primera aproximación al diseño de la guarida. Sin embargo fue desecharlo por la estética cavernosa. Finalmente se llega al diseño de la figura 5.13, que en lugar de parecer una mina o una cueva, se inspira el aspecto de un paisaje urbano, y los colores se adaptan al resto de la interfaz.

Las salas de la guarida mantienen una estética similar, sin embargo contienen diversos colores y formas (no armónicas entre si). Esto es para dar a la guarida el aspecto urbano deseado (generalmente las fachadas de los establecimientos de una caye son bastante distintos y están desordenados), además así se diferencian a simple vista (esto es importante para poder identificar cada sala sin problemas).

Aunque solo se ve la puerta de cada sala, cuando se hace *click* en la imagen de alguna de ellas, se abre otra ventana inferior con los detalles de la sala, acciones



Figura 5.13: Vista final de la guarida. Se cambia el aspecto cavernoso por otro más urbano.

relacionadas y un dibujo esquemático que refuerza el concepto y utilidad de la misma.

5.3.4. Representación gráfica de los monstruos

Es importante que los diseños de las razas de monstruo sean diferentes e individualmente originales, pero manteniendo la coherencia y homogeneidad con el resto del juego. Desde el principio estaba claro el tipo de monstruos que se querían hacer. La idea principal consistía en que los monstruos no debían dar miedo, pero sí mantener cierto grado de rareza, siendo algo paranoicos (estilo . Los jugadores deben sorprenderse al ver los diseños de las nuevas razas desbloqueadas para mantener el deseo de conocer más, colecciónar y seguir jugando. Por otro lado, se evita todo lo posible el estilo japonés, no deben parecer *pokemons* o *digimons*, porque este juego es para otro tipo de público. En definitiva, siempre se buscaron diseños sencillos, fáciles de recordar y con personalidad propia.

En la figura 5.14 se pueden ver algunos bocetos previos, cuyo estilo se fue adaptando hasta llegar al diseño actual. Tuvieron que hacerse muchos dibujos diferentes para luego poder seleccionar aquellos más representativos. También hubo que hacer varias pruebas de dibujo (a mano alzada, con lápices de colores, con efectos digitales, dibujos escaneados, fotos de muñecos hechos en plastilina, etc.) para poder tener claro el método de dibujo en los gráficos de los monstruos, manteniendo el equilibrio entre la calidad del resultado final y el tiempo necesario para desarrollarlo.

Al final, de todas las razas plasmadas en los bocetos, se eligieron solamente ocho, y de todos los métodos de dibujo probados, se utilizaron gráficos vectoriales creados con el programa *Inkscape* [2]. Para colorear los monstruos se utilizan texturas inspiradas en las nuevas colecciones de varias marcas de ropa conocidas, logrando una sensación fresca y joven. La solución es relativamente sencilla: primero se dibuja el monstruo en papel, luego se escanea y se importa con el *Inkscape*, donde se crea el gráfico vectorial, y finalmente se exporta el mapa de bits (al formato PNG) para que pueda ser mostrado en los navegadores de los

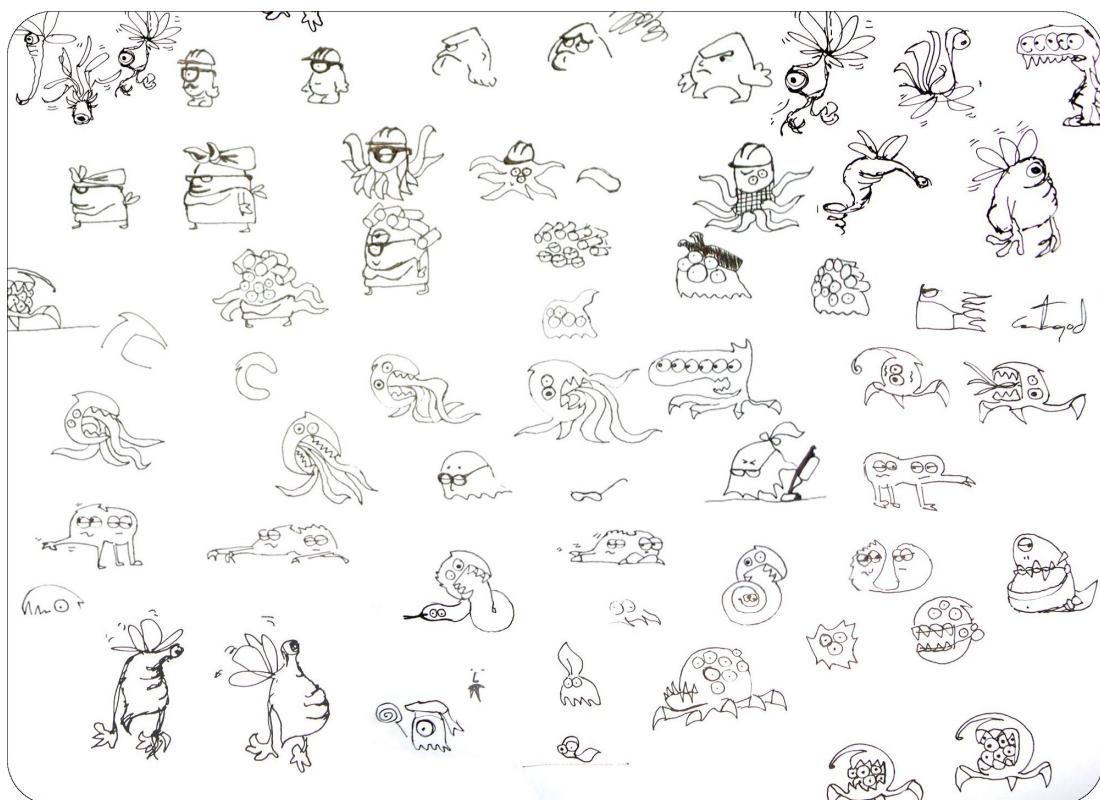


Figura 5.14: Algunos bocetos previos realizados para idear nuevas razas de monstruo.



Figura 5.15: Evolución del diseño y estilo gráfico de la raza de monstruo *Quad*.

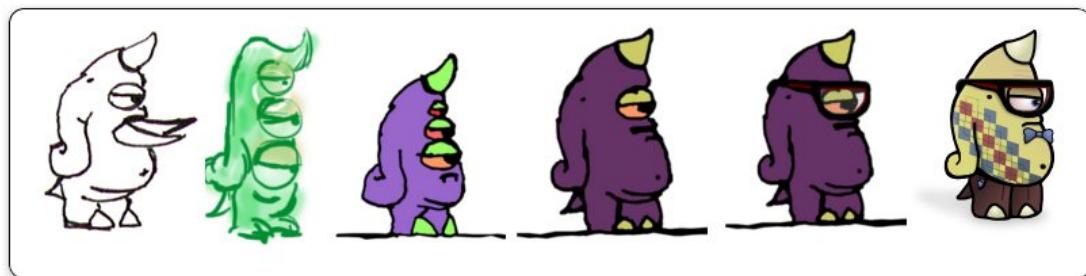


Figura 5.16: Evolución del diseño y estilo gráfico de la raza monstruo *Ubunto*.



Figura 5.17: Proceso para crear los gráficos de los monstruos. A partir de un dibujo escaneado se crea el modelo vectorial, que después se convierte en mapa de bits para poder ser integrado en la página web.

usuarios. El tiempo medio para dibujar un monstruo con este método es de unas cinco horas, y el resultado es excelente. En la imagen de la figura 5.17 se pueden ver algunos ejemplos de como se corresponde el dibujo hecho a mano con el gráfico final.

5.3.4.1. Razas de monstruo

Cada raza de monstruo tiene carácter diferente dependiendo de la sensación que se quiera provocar en el jugador. No deben tener el mismo aspecto una raza que aparece al comienzo del juego y otra que ha sido muy difícil de conseguir. Por ejemplo, el diseño de la primera raza de monstruo que aparece en el juego parece débil y vulnerable, sin embargo expresa ilusión y alegría, que son las sensaciones que debería tener el jugador al principio. En la figura 5.17 se pueden ver los ejemplos de la raza *Bu* (monstruos verdes) y de la raza *Polbo* (monstruos azules), que son las dos primeras razas que posee el jugador.

A modo de ejemplo, se comparan los gráficos de las siguientes razas con sus

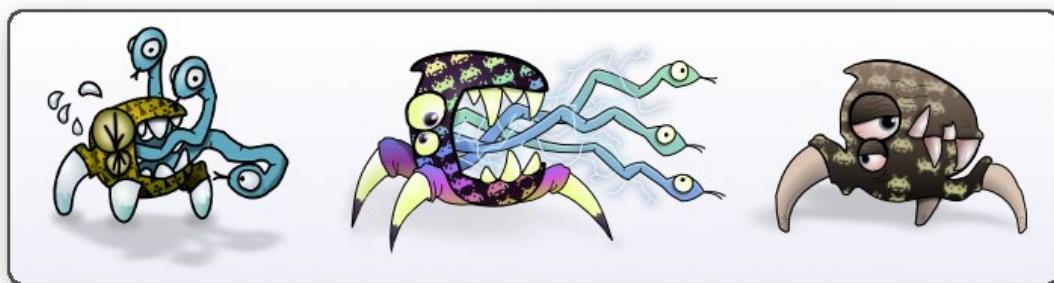


Figura 5.18: Cría, adulto y anciano de la raza de monstruo *Electroserpe*. Se trata de monstruos peligrosos y efectivos en combate cuando desarrollan sus habilidades eléctricas.

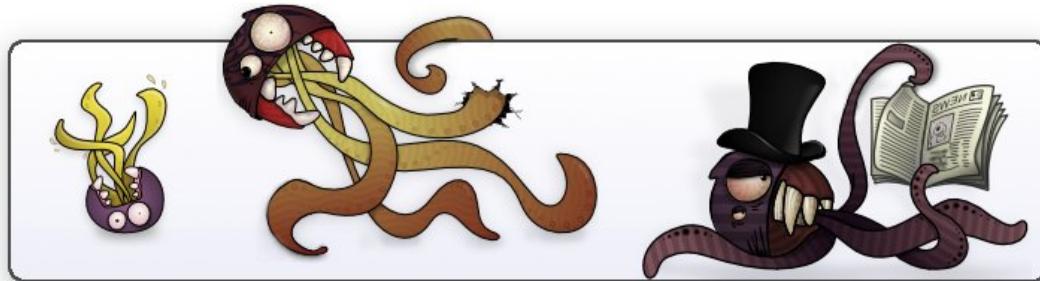


Figura 5.19: Cría, adulto y anciano de la raza de monstruo *Quad*. Son los monstruos con más fuerza que hay en el juego.



Figura 5.20: Cría, adulto y anciano de la raza de monstruo *Ubunto*. Es la raza con más inteligencia.

características en el juego:

- **Electroserpe:** Ya desde pequeños son rebeldes, caprichosos y siempre están llorando, lo cual es coherente con el hecho de que los monstruos más longevos como los de esta raza requieren más cuidados cuando son crías para mejorar todo lo posible sus atributos antes de ser adultos. Cuando son mayores tienen un aspecto *retro* descerebrado que mezcla los conceptos “serpiente”, “electricidad” y “locura” para mantener firme la idea de una raza de monstruos no muy inteligentes, fuertes y con la capacidad de desarrollar habilidades de combate muy peligrosas. La textura *space invaders* encaja de lleno con los conceptos anteriores, que pierden fuerza cuando envejecen porque ya no son tan peligrosos.
- **Quad:** Son los monstruos más fuertes del juego, útiles tanto para batallas como para trabajar en la guarida. Tampoco son muy inteligentes, aunque mejoran a los *Electroserpe*. El dibujo que representa a los adultos demuestra su fuerza rompiendo el propio marco donde se exponen los monstruos. El diseño del *Quad* es como un calamar gigante (de esos que hunden barcos en la literatura popular), cuyos tentáculos (lenguas) terminan en una peligrosa boca con dientes. Las lenguas están coloreadas con tonos amarillos y rojos como si fuesen “fuego”, de esta manera también se parece a un dragón (que también simboliza fuerza). Sus ojos son rojizos y están descontrolados, su piel oscura está decorada con una especie de tatuaje tribal y las encías le salen de la boca de forma exagerada (síbolizando un carácter agresivo y salvaje). Cuando envejecen, el nivel de agresividad disminuye notablemente porque a esa edad ya puede dedicarse a leer y a pasar el tiempo que les queda, recordando al jugador que un monstruo anciano ya no puede luchar en las misiones ni ayudar en las tareas domésticas, pero sí que puede para cuidar y enseñar habilidades a las nuevas generaciones.
- **Ubunto:** Esta raza es la mejor de todas, por eso es la última en ser desbloqueada. Estos monstruos son más caros, más difíciles de cuidar y los

tiempos de incubación y metamorfosis son más altos de lo habitual. Tienen mucha inteligencia y una larga esperanza de vida, lo cual les permite desarrollar las habilidades hasta niveles muy altos. Con esta raza va a ser mucho más importante cuidar la calidad del sistema de enseñanza en la guarida. Será posible conseguir monstruos de muy buena calidad, pero también va a ser más difícil. Las crías son como niños que siempre están pensando en jugar, gorditos y bien cuidados, porque se supone que a estas alturas el jugador ya dispone de buenas instalaciones en general. Los adultos expresan soberbia porque se sienten importantes y mejores que los demás monstruos. Hasta el propio nombre *Ubunto* viene del sistema operativo libre “Ubuntu”, que se relaciona con gente lista que “sabe de informática” (hablando de círculos ajenos a la informática, ya que dentro todos sabemos que instalar y utilizar *Ubuntu* es realmente sencillo), y el color del monstruo también trata de relacionarlo con este concepto. En general, el aspecto del monstruo está inspirado en la forma de vestir de los *Nerd*, que es como se les llama en los Estados Unidos a los “empollones de la universidad”. Los *Ubunto* ancianos son casi iguales que los adultos, mantienen el mismo estilo y la misma ropa pero con barba, de este modo se enfatiza lo obstinados que son, ya que ellos están convencidos de que son superiores y mejores, y nunca cambiarán de idea.

Capítulo 6

Conclusiones

Índice general

6.1. Resultados obtenidos	171
6.2. Trabajo Futuro	173

6.1. Resultados obtenidos

Como resultado de cumplir los requisitos establecidos se ha implementado una aplicación Web en la cual se desarrolla el juego *Thearsmonsters*. Esta aplicación es perfectamente funcional y cumple todos los objetivos establecidos. Con respecto al juego, solamente se han añadido las características mínimas para que se pueda jugar ya que sería inviable incluirlas todas (a partir de cierto punto, añadir más contenido es una tarea mecánica que necesita mucho trabajo).

La aplicación Web desarrollada en este proyecto ya está lista para funcionar en producción. Es posible generar un paquete *war* para que se pueda desplegar en un servidor hospedado en Internet.

Las características más destacadas de la aplicación son las siguientes:

- **Dentro de las tendencias Web 2.0:** En *Thearsmonsters* la interacción entre los usuarios es una de las partes más importantes. El soporte para

acciones *Ajax* permite una mejor interacción entre el usuario y la aplicación. Además existe un buen soporte para incluir servicios Web según vayan haciendo falta.

- **Estabilidad:** El modelo de la aplicación está meticulosamente validado con un conjunto de pruebas unitarias.
- **Mantenible:** El diseño y la implementación se han realizado aplicando las mejores prácticas, siempre aplicando los patrones de diseño adecuados a cada situación. La vista se divide en componentes reutilizables. La base de datos puede ser reemplazada sin tener que realizar demasiado esfuerzo.
- **Eficiencia:** Se ha minimizado el acceso a la base de datos prácticamente a una o dos consultas por cada caso de uso. También se hace caché de los datos que son necesarios constantemente en el controlador.
- **Internacionalización:** Traducir la aplicación a otros idiomas es muy sencillo porque el código está preparado para ello. Lo único necesario es traducir un simple fichero de texto llamado `Messages.properties`.
- **Seguridad:** Los usuarios son identificados en todas las acciones, que además se realizan en términos de *transacciones*. Las contraseñas se guardan cifradas en la base de datos.
- **Escalabilidad:** Todos los objetos que necesitan ser guardados en la sesión son *serializables* y se mantienen las restricciones necesarias para que la aplicación pueda ser desplegada en un *clúster* de servidores (como la abstención del uso de variables globales).

La implementación se hace en *Struts*. Al tratarse de un *framework* muy conocido y utilizado, resultaría fácil encontrar más profesionales que puedan incorporarse al proyecto. Sin embargo otros *frameworks* más modernos como *Ruby on Rails* [7] o *Django* [5] serían una mejor opción en el caso de este proyecto

porque están basados en el *desarrollo ágil* y permiten realizar la implementación en tiempos más reducidos, con muchas menos líneas de código.

Una de las características más importantes para un juego multijugador masivo es la escalabilidad. Habría sido una muy buena idea desarrollar la aplicación sobre el paradigma *Cloud Computing* [35], por ejemplo, con la versión limitada del *framework Django* que se puede desplegar sobre la nube de *Google App Engine* [37].

Entre otras cosas se ha aprendido que hacer un juego de computadora es un trabajo realmente complicado. Es necesario combinar el arte del diseño gráfico con la técnica de la informática [24], habilidades que resulta difícil encontrar en la misma persona. El desarrollo de la interfaz gráfica es una tarea mucho más exigente de lo esperado, así como el diseño del mundo persistente, que debe estar equilibrado y ser perfectamente estable.

El desarrollo y perfeccionamiento de una idea original supone muchísimo trabajo adicional. Si el juego *Thearsmonsters* se hiciese, al igual que la mayoría de juegos de este tipo, al estilo *Ogame* y con un diseño similar al que tienen los demás portales *Web* (dedicando menos tiempo al apartado artístico), el esfuerzo requerido hubiera sido por lo menos tres o cuatro veces menor. Aunque bien es cierto que la originalidad es en muchos casos la única posibilidad de éxito en un sector tan saturado y lleno de “clones” como es el de los juegos multijugador masivos para navegador. La próxima vez, al menos se tendrá en cuenta este factor a la hora de realizar la planificación.

6.2. Trabajo Futuro

El juego *Thearsmonsters* ya ha sido concebido como una aplicación que evoluciona. El modelo conceptual es muy extensible, sobre él se pueden añadir fácilmente más elementos para enriquecer la experiencia de juego sin alterar el equilibrio existente. Conociendo la necesidad de incorporar nuevos contenidos en el futuro, toda la aplicación ha sido diseñada para ser fácilmente extensible. Por

ejemplo los tipos de sala se diferencian básicamente en una serie de atributos, así cuando haya que añadir una sala nueva casi es suficiente con darle un valor a estos atributos y ya está.

El reto más importante al que se tendrá que enfrentar la aplicación es el despliegue en producción con usuarios reales. Para que esto suceda, en primer lugar la vista deberá ser mejorada para que se pueda visualizar correctamente en más navegadores. Actualmente ha sido validada en *Firefox* y *Opera* pero es muy importante que funcione sobre *Microsoft Internet Explorer* (el más utilizado en la red). También será necesario validar que el despliegue de la aplicación sobre un *clúster* de servidores funciona correctamente.

Con respecto al juego, las próximas expansiones que están planeadas son:

1. **Mejorar la página principal** del sitio Web añadiendo contenido extra, capturas de pantalla, noticias, etc. Así se podrá captar la atención de más jugadores.
2. **Wiki con el manual de usuario** accesible desde cualquier punto del juego.
3. **Foro de la comunidad** de jugadores donde los usuarios del juego puedan compartir sus experiencias.
4. **Buscador de salas públicas** y puesta en marcha de la publicación de las salas (implementado parcialmente dentro del ámbito de este proyecto).
5. **Sistema de mensajería** entre los jugadores. Poder escribir comentarios en la guarida de otros jugadores y enviar correos privados.
6. **Sistema de misiones** para ir desbloqueando razas y salas. Esto requiere también implementar el sistema de batallas entre monstruos.
7. **Ranking** con los puntos de cada jugador.
8. **Alianzas de jugadores** para que puedan formar grupos sociales y hacer misiones multijugador.

9. **Armas y objetos** que doten de mejores características a los monstruos que los poseen.
10. **Subasta** para compra/venta de objetos y monstruos.
11. **Torneos PvP** para que los jugadores puedan participar en eventos organizados y medir la fuerza de combate de sus monstruos.

La mayoría de estas ampliaciones ya están contempladas en el capítulo de análisis, y gracias a ello la implementación actual ya está preparada para ser actualizada con estas ideas de manera natural.

Actualmente el juego solamente puede ser visualizado de manera cómoda y adaptada sobre un navegador Web. En el futuro será muy conveniente para aumentar considerablemente el número de jugadores, añadir nuevas vistas ajustadas para dar soporte y acceso a jugadores que se conecten desde otras plataformas: móviles (WAP), *iPhone*, televisión digital interactiva, etc. Gracias a la estructura en capas de la aplicación, para realizar esto solamente habrá que cambiar la parte dedicada a la presentación.

Apéndice A

Elección del *framework* Web

En general, con el término *framework*, nos estamos refiriendo a una estructura software compuesta de componentes personalizables e intercambiables para el desarrollo de una aplicación. Se puede considerar como una aplicación genérica incompleta y configurable a la que podemos añadirle las últimas piezas para construir una aplicación concreta [34].

Un *framework* Web, por tanto, podemos definirlo como un conjunto de componentes (por ejemplo clases en *Java*, descriptores, archivos de configuración en XML, etc.) que componen un diseño reutilizable que facilita y agiliza el desarrollo de sistemas Web.

Una aplicación Web también se puede desarrollar sin la necesidad de utilizar un *framework*, pero no es recomendable. En general, la principal desventaja de utilizar un *framework* es la curva de aprendizaje ya que normalmente cuesta mucho esfuerzo acostumbrarse a utilizarlo, superar los problemas técnicos, adaptar el *framework* al domino y adaptarlo al problema concreto. Sin embargo, una vez superada la curva de aprendizaje, la productividad, escalabilidad, portabilidad y el mantenimiento de la aplicación son mucho mejores.

La mayoría de *frameworks* Web se construyen en base al patrón arquitectónico MVC (Modelo-Vista-Controlador [32]). En la figura A.1 se puede ver un ejemplo de cómo se implementa esta estructura en una aplicación Web *Java* común, que consiste, a grandes rasgos, en la utilización de *servlets* para procesar las peti-

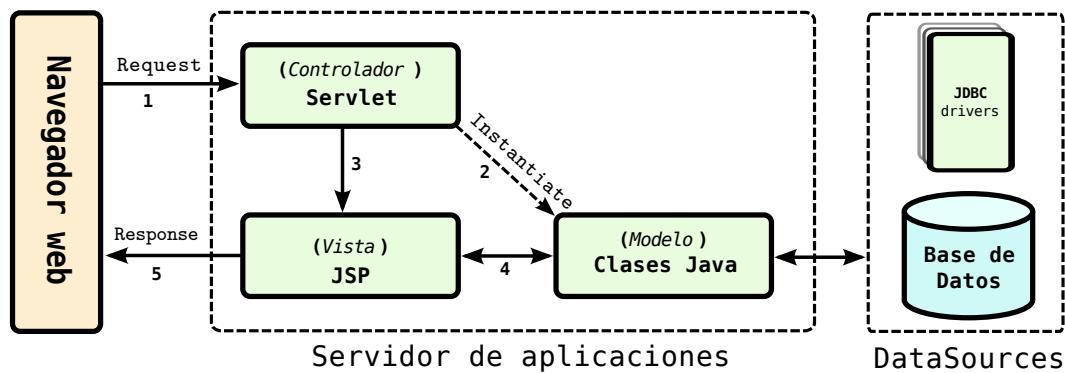


Figura A.1: Ejemplo de implementación del patrón arquitectónico MVC en *Java*.

ciones (controladores) y páginas JSP para mostrar la interfaz de usuario (vistas), implementando la parte del modelo mediante *JavaBeans* o *POJOs*.

Existen muchos *frameworks* Web para elegir. La elección de uno u otro depende de las características de cada problema y de los componentes del equipo, aunque dentro del mismo problema también existen varias alternativas. A continuación se muestran las características más importantes de algunos de los *frameworks* más conocidos:

- **Ruby on Rails**: También conocido como *Rails* o *RoR*, está escrito en el lenguaje de programación dinámico *Ruby*. Lo más importante de *Rails* es su filosofía (casi todos los nuevos *frameworks* Web que van saliendo están inspirados, al menos en parte, en lo que se llama *el estilo Rails*). Entre los principios fundamentales de *Rails* se encuentran *DRY* (del inglés *Don't repeat yourself*, que significa que las definiciones deberían hacerse una sola vez) y *convención sobre configuración* (que el programador sólo necesite definir aquella configuración que no es convencional). [7]
- **Django**: Escrito en *Python*. Al igual que *Rails*, una vez que se conoce bien el *framework* resulta muy productivo. Tiene su propio estilo, pero también está basado en el *desarrollo rápido de aplicaciones*. [5]

- **Struts:** Funciona sobre el estándar *Java Servlet API*, de la especificación *Java EE*. Este *framework* es algo antiguo, por lo tanto su filosofía puede estar un poco desfasada, pero su estabilidad, escalabilidad y utilidad están aseguradas. La madurez de *Struts* es una de sus principales ventajas, además soporta la integración de otras herramientas como *Hibernate* o *Spring*. [4]
- **Tapestry:** Al igual que *Struts*, complementa y construye desde el estándar *Java Servlet API*, funcionando también en cualquier servidor contenedor de *servlets* o contenedor de aplicaciones. Sin embargo propone una perspectiva más fresca y sencilla que *Struts* para el desarrollo de los componentes de la aplicación.
- **Microsoft .Net:** Es un componente de software que puede ser añadido al sistema operativo *Windows*, que es el producto principal en la oferta de *Microsoft* para ser utilizada por la mayoría de las aplicaciones creadas para este sistema operativo. Lo bueno de .NET es que incluye todo lo necesario para desarrollar, por lo tanto el programador no tiene que preocuparse tanto por la integración entre los distintos componentes. Lo malo es que diferencia de la mayoría de *frameworks* disponibles no es independiente de la plataforma, y además no es libre.
- **Cake PHP:** Escrito en *PHP* y creado sobre los conceptos de *Ruby on Rails*. Al igual que éste se facilita al usuario la interacción con la base de datos mediante el uso de *ActiveRecord*. El lenguaje de programación *PHP* es muy fácil de entender y de utilizar, pero también tiene muchas características que lo hacen poco recomendable (entre otras solo tiene soporte para objetos en las últimas versiones y en general no está muy enfocado desde el punto de vista de la ingeniería).

En general, los *frameworks* más actuales son más intuitivos y resuelven los problemas clásicos con más soltura y elegancia, sin embargo están en constante cambio y resulta difícil mantener el código actualizado.

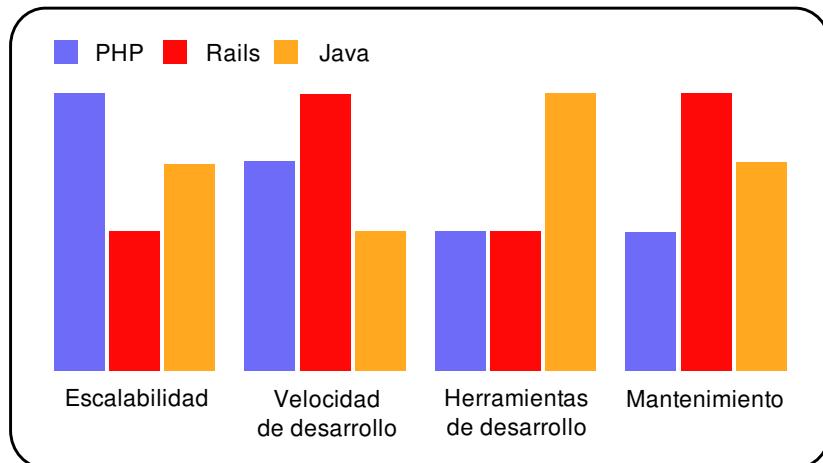


Figura A.2: Comparación entre los tres grandes sectores del desarrollo Web libre, expuesta por *Tim Bray* en la *PHP International Conference* en el año 2006.

La elección de un *framework* es casi la parte más importante de un proyecto, ya que de esta decisión dependen muchos de los aspectos relacionados con el diseño y con la implementación. Existen infinidad de comparaciones, opiniones, artículos, pero al final siempre depende del caso particular (especialmente de los conocimientos previos del equipo de desarrollo). Por ejemplo, una de las tantas comparaciones entre alternativas es la que realizó *Tim Bray* en la conferencia internacional de PHP en el 2006 (figura A.2), dando valores aproximados a los entornos más generales y utilizados dentro de cuatro categorías diferentes. Las comparaciones son exclusivas al ámbito del desarrollo Web, por ejemplo, con respecto a la escalabilidad aparece que PHP es mejor que *Java*, lo cual según el propio *Tim* no es cierto, sin embargo lo pone así porque conseguir software escalable con PHP es mucho más sencillo¹.

En el capítulo de diseño (apartado 4.4.1) se ha comentado que la aplicación Web *Thearsmonsters* se desarrolla con el *framework Struts*. Para tomar esta decisión se han seguido los siguientes criterios:

¹La gráfica de la figura A.2 es parte de un artículo en su blog personal en <http://www.tbray.org/ongoing/When/200x/2006/11/10/Comparing-Frameworks>

- Aprender a manejar un nuevo *framework* puede llevar demasiado tiempo, sobre todo si hay que hacerlo sin el soporte o la ayuda de alguien que lo conozca a fondo. Los problemas técnicos derivados de la configuración, versiones del producto, instalación o errores del propio *framework* pueden retrasar peligrosamente el desarrollo de la aplicación.
- El proyectando, en el momento de comenzar el proyecto, ya tenía una buena base en los *frameworks* *Struts* y *.NET* (gracias a la asignatura de 5º *Integración de Sistemas*). Además se siente cómodo con el lenguaje de programación *Java* (debido a otras múltiples asignaturas). El framework *.NET* fue inmediatamente descartado por su elevado coste (*Struts* en cambio es gratuito).
- También se tenían conocimientos de *PHP* (gracias a las prácticas realizadas en la empresa *Interacción* durante el verano del 2007), sin embargo no se sabía utilizar ningún *Framework* Web (como *CakePHP*, *Symfony* o *Zend*) excepto *Drupal*, pero este CMS no sirve para implementar el juego *Thearsmonsters*.
- La utilización de *Struts* no supone para nada el incumplimiento de alguno de los objetivos establecidos en la fase análisis, y a pesar de sus desventajas sigue siendo una solución estable y razonable.

Aunque cabe destacar que si no fuese por la carga que supone la curva de aprendizaje, la elección tomada habría sido diferente. Dadas las características de la aplicación Web *Thearsmonsters*, una mejor solución sería utilizar el *framework Ruby on Rails*² o *Django*. El tiempo de desarrollo, para un programador experto en todos los *frameworks* nombrados, en este caso, sería mucho más reducido al utilizar *Rails* (figura A.2) o *Django*, y la calidad del software seguiría

²Durante el desarrollo del proyecto, el proyectando ha adquirido seis meses de experiencia con *Rails*, pero ya no compensa la migración desde *Struts* porque habría que reescribir casi toda la aplicación.

siendo potencialmente elevada. Además el despliegue de aplicaciones desarrolladas con estas últimas tecnologías ya tiene muy buena salida, por ejemplo, tanto *Django* como *Rails* (este último sobre *JRuby*) se pueden desplegar (con ciertas restricciones) en la nube *Google AppEngine* [37] (no se van a comentar las ventajas del *Cloud Computing* [35], pero sí que se puede afirmar que para el despliegue de un juego masivo multijugador este paradigma resulta ideal).

Apéndice B

Breve introducción a Ajax

La mayor parte de las interacciones del usuario causan una petición HTTP al servidor Web, que la procesa y devuelve la nueva página a mostrar. Internamente la petición ejecuta una lógica de negocio (por ejemplo: comprar un huevo, ver monstruos de la guarida, asignar una tarea, etc.) y devuelve una nueva página HTML con la respuesta (también puede ser un error o una redirección). Este modelo, cuya sencillez se puede apreciar en la figura B.1, se denomina “modelo de aplicaciones Web clásico”. [23]

El modelo clásico ha sido y es útil, está soportado por numerosas tecnologías maduras y es más fácil de desarrollar, sin embargo las interfaces de usuario son poco interactivas con respecto a las aplicaciones de escritorio, no tienen soporte para *drag-n-drop*, no es posible refrescar la información de una zona de la página sin tener que recargar toda la página, etc.

Ajax significa *Asynchronous JavaScript + XML*, aunque realmente no es necesario el uso de XML (la respuesta del servidor puede ser JSON, texto plano, HTML . . .). Es una técnica popularizada por *Google* (*Google Maps*, *Gmail*, *Google Calendar*, etc.) que presenta un enfoque para la construcción de aplicaciones Web con una interactividad similar a la que presentan las aplicaciones de escritorio. La parte cliente es en su mayor parte *JavaScript* y la parte servidora recibe peticiones HTTP asíncronas procedentes de la parte cliente. Como se ve en la figura B.2, este modelo es más complicado que el clásico, sobretodo por el hecho de que

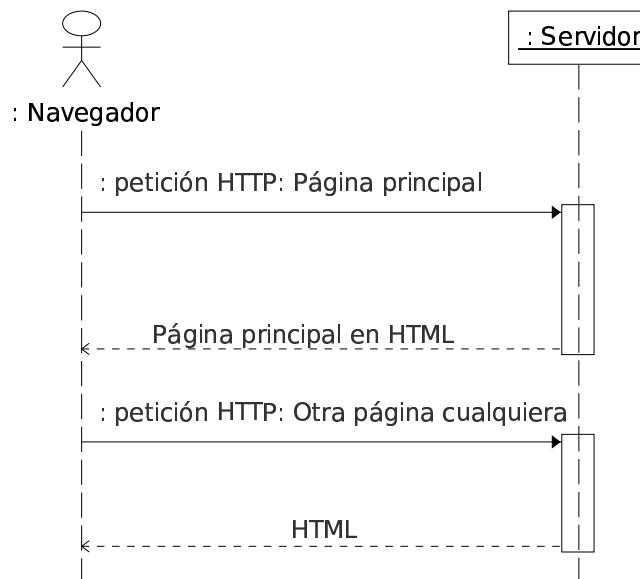


Figura B.1: Petición HTTP en el modelo de aplicaciones Web clásico.

necesita el uso de *JavaScript* para ser implementado, código que se ejecuta en el navegador, que causa problemas de portabilidad y es muy propenso a errores.

El principal inconveniente del enfoque *Ajax* es la dificultad de desarrollo: *frameworks* poco maduros (cambiantes), tediosos de usar (bajo nivel) y normalmente requieren escribir código *JavaScript* (gran esfuerzo para garantizar que funcione en todos los navegadores). Por ese motivo actualmente es más prudente usar *Ajax* sólo para implementar casos de uso que verdaderamente lo requieren.

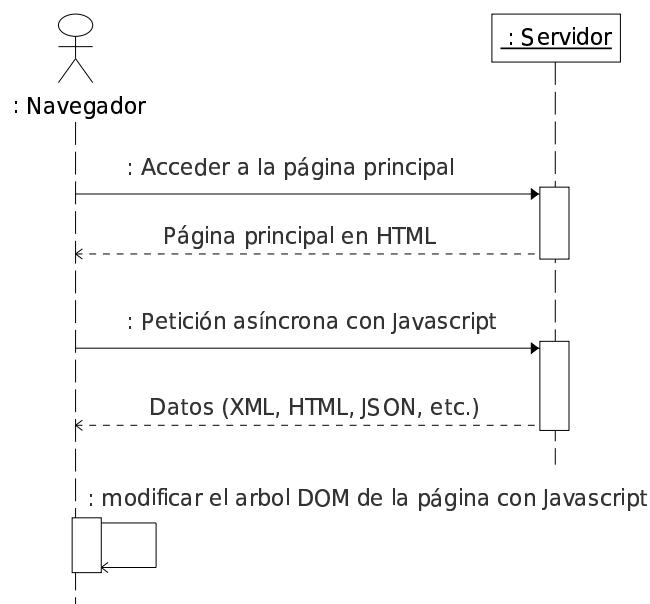


Figura B.2: Petición HTTP con el modelo de aplicaciones Web *Ajax*. Requiere el uso de *JavaScript* en el navegador.

Apéndice C

Juegos MMORPG para navegador de referencia

Uno de los primeros MMORPG basados en navegador es *Archmage*, que se remonta a principios de 1999. El jugador tomaba el rol de un mago de la edad media con poderes arcanos, e intentaba dominar una porción de territorio, pero este proyecto se fue al traste por la gravedad de un *bug* que la empresa desarrolladora nunca consiguió corregir además de los continuos ataques de *hackers* que recibía.

Actualmente, algunos MMORPGs basados en navegador como *Kings of Chaos*, contienen una población de jugadores de centenares de miles. Otros ejemplos de juegos de este estilo son *Legend of the Green Dragon* y *World of Chaos*, que son software libre, lo que permite que cualquiera pueda crear un servidor con su propio juego. En *BattleMaster* el mundo nunca se reinicia, de forma que algunos reinos de jugadores llevan existiendo más de cinco años reales mientras que otros reinos se han creado y han sido destruidos en ese mismo periodo de tiempo, desarrollando su propia historia virtual, vivida y escrita por los propios jugadores. Cada juego cuenta con sus características particulares.

Agrupando los juegos existentes por la temática que siguen es fácil observar hay poca innovación, ya que a penas se diferencian unas pocas categorías:

- **Juegos espaciales**, donde hay naves, planetas y galaxias que dominar.

Podemos encontrar títulos como *Ogame*, *AstroWars*, *X-Wars*, *Space 4k*, *Astratega*, *Virtual Galaxy*, *Dark Galaxy*, *Colony Wars*, *Ferion*, *Pardus*, *Star Wars Combine*, etc.

- **Juegos medievales fantásticos**, donde puede haber caballeros, orcos, elfos, espadas y magos. Encontraremos títulos como *Travian*, *Empire Strike*, *The reincarnation*, *The Five Pilars*, *Dark Throne*, *Kings of Chaos*, *Legend of the Green Dragon*, etc.
- **Zombies, hombres lobo, vampiros** y otros personajes de ciencia ficción como *Monsters game*, *Bite Fight*, *Urban Death*, etc.
- También se abordan otros temas, por ejemplo *Vendetta* trata sobre la mafia, *Earth 2025* sobre un mundo futuro en guerra, *Dig'nFight* es una versión sencilla para navegador del clásico *Dungeon Keeper*, *Hogwarts Live* trata sobre Harry Potter, en *Popomundo* cada personaje tiene que llegar a ser una estrella de rock, en *Hattrick* hay que gestionar un equipo de fútbol y en *Nation States* hay que gestionar un país.

Como vemos, la mayoría de títulos encajan en la misma temática. Pero la originalidad es incluso menos de lo que parece, porque la forma de jugar es casi siempre la misma. Por ejemplo en *Ogame* y *Vendetta*, que tratan sobre el espacio y la mafia respectivamente, hay una equivalencia evidente entre planetas y edificios o entre naves y matones, es decir que en realidad el juego es el mismo pero con otro aspecto. Si agrupásemos los juegos anteriormente mencionados por estilo de juego tendríamos que pueden ser de hacer ejércitos y atacar a los demás, de hacerse un único personaje y mejorarlo para atacar a los demás, o bien de gestionar económicamente algún tipo de entidad (como un equipo de fútbol o de baloncesto), es decir que tampoco hay mucha variedad en este aspecto.

Hay listas con enlaces a todos los juegos mencionados en la *Wikipedia* (<http://es.wikipedia.org/wiki/M> http://en.wikipedia.org/wiki/List_of_MMORPGs), en *Massivemultiplayer* (<http://www.massivemultipl>) y en artículos de *Blogspot* (<http://mmogs.blogspot.com/2006/05/la-lista-de-brbmmogs.html>).

Apéndice D

Manual para conectarse y utilizar la aplicación

En el CD adjunto se incluye el código fuente por si fuese necesario examinarlo. También se incluye documentación y software suficientes para poder ejecutar la aplicación por cuenta propia. Pero este proceso puede resultar largo y tedioso (como mínimo hay que instalar *MySQL*, *Tomcat* y sus dependencias, y después configurar todo correctamente para que funcione la aplicación). Por ello se habilita una URL con la aplicación ya funcionando, accesible desde Internet.

Por lo tanto, lo único que hay que hacer es:

1. Conectarse a la URL <http://bulma.dc.fi.udc.es:8080/theearsmonsters>
2. Pulsar sobre la pestaña **registrarse**
3. Introducir los datos requeridos para crear una cuenta
4. Identificarse en el formulario de *login* y ¡a disfrutar!

Apéndice E

Contenidos del CD

El CD adjunto incluye:

- El código fuente de la aplicación.
- *Software* y documentación para reproducir el entorno de desarrollo o de producción de la versión actual de la aplicación. Todo el *software* incluido se puede instalar tanto en *Linux* como en *Windows*.

Apéndice F

Glosario de acrónimos

3D *3 Dimensiones.*

AJAX *Asynchronous JavaScript + XML.*

API *Application Programming Interface.*

CMS *Content Management System.*

DAO *Data Access Object.*

DOM *Document Object Model.*

DRY *Don't Repeat Yourself.*

DWR *Direct Web Remoting.*

IE *Internet Explorer.*

Java EE *Java Platform, Enterprise Edition.*

JSTL *Java Standard Tag Library.*

MMOG *Massively Multiplayer Online Game.*

MMORPG *Massively Multiplayer Online Role Player Game.*

MVC *Model View Controller.*

ORM *Object-Relational Mapping.*

PC *Personal Computer.*

PNJ *Personaje No Jugador (en inglés NPC).*

PvP *Player versus Player.*

RGB *Red Green Blue.*

RPG *Role Player Game.*

TLD *Tag Library Descriptor.*

UML *Unified Modeling Language.*

URL *Uniform Resource Locator.*

VO *Value Object.*

WoW *World of Warcraft.*

Apéndice G

Glosario de términos

Bitmap Imagen digital rasterizada. En general es una matriz de píxeles de colores aplicando algún tipo de compresión.

Casual player Tipo de jugador que juega de vez en cuando, sin mucha presión y no es muy experto.

Checkbox En una interfaz gráfica, es una lista de elementos con cuadrados a la izquierda de cada elemento para que se pueda seleccionar.

Driver También llamado *controlador*, es un paquete informático que permite al sistema operativo (o a otro cliente de cualquier tipo) interactuar con un *periférico*, haciendo una abstracción del *hardware* (o de otra capa *software* de más bajo nivel) y proporcionando una *interfaz* para poder usarlo.

Estándar abierto Es una especificación disponible públicamente para lograr una tarea específica. Es lo contrario a un estándar con patentes (no abierto), que puede imponer sobrecargos u otros términos de licencia en las implementaciones del estándar.

Hardcore player Jugador compulsivo que utiliza muchas horas del día para jugar. Es muy competitivo y difícil de derrotar en juegos multijugador.

Internauta Usuario de la *World Wide Web*, se utiliza este término por la expresión “navegar por Internet”, que significa visitar una serie de páginas web utilizando un navegador Web.

Noob player Jugador novato. Son muy típicos de los juegos en línea y se consideran torpes e inexpertos.

Píxel Menor unidad homogénea en color que forma parte de una imagen digital, ya sea esta una fotografía, un fotograma de vídeo o un gráfico.

Uber Guild Alianza de jugadores con pocos componentes pero de muy alto rango o nivel.

Web semántica Se basa en la idea de añadir metadatos semánticos y ontológicos a la *World Wide Web*. Esas informaciones adicionales (que describen el contenido, el significado y la relación de los datos) se deben proporcionar de manera formal, para que así sea posible evaluarlas automáticamente por máquinas de procesamiento. El objetivo es mejorar *Internet* ampliando la interoperabilidad entre los sistemas informáticos y reducir la necesaria mediación de operadores humanos.

World Wide Web También llamada *Red Global Mundial* es un sistema de documentos de hipertexto y/o hipermedios enlazados y accesibles a través de *Internet*. Con un navegador Web, un usuario visualiza páginas web que pueden contener texto, imágenes, vídeos u otros contenidos multimedia, y navega a través de ellas usando hiperenlaces.

Zerg Guild Alianza de jugadores de poco rango que vuelve competitiva debido a su gran número de componentes.

Bibliografía

- [1] Página principal del gimp.
<http://www.gimp.org>.
- [2] Página principal del inkscape.
<http://www.inkscape.org>.
- [3] Página principal del photoshop.
<http://www.adobe.com/products/photoshop>.
- [4] Página principal del proyecto *Apache Struts*.
<http://struts.apache.org>.
- [5] Página principal del proyecto *Django*.
<http://www.djangoproject.com>.
- [6] Página principal del proyecto *JQuery*.
<http://jquery.com>.
- [7] Página principal del proyecto *Ruby on Rails*.
<http://rubyonrails.org>.
- [8] Diseño de iconos, errores típicos a evitar.
<http://turbomilk.com/truestories/cookbook/criticism/10-mistakes-in-icon-design>, 2008.
- [9] Recopilación de buenas portadas para sitios web.
<http://coolhomepages.com>, 2008.

- [10] Tutoriales y recursos para la creación de videojuegos.
<http://www.coldstoragedesigns.com>, 2008.
- [11] Galería de imágenes con diferentes diseños web, divididos en diferentes categorías.
<http://www.flickr.com/photos/guspim/collections/72157600047307884>, 2009.
- [12] Librería con patrones de diseño de interfaces gráficas.
<http://ui-patterns.com>, 2009.
- [13] Patrones de diseño gráfico en sitios web que utilicen tecnologías ajax.
<http://ajaxpatterns.org>, 2009.
- [14] D. Alur, J. Crupi, and D. Malks.
Core J2EE Patterns: Best Practices and Design Strategies.
Prentice Hall / Sun Microsystems Press, June 2001.
- [15] J. Cantero.
Analítica y estudio del juego *Ogame*.
http://kanterobi.blogspot.com/2006/01/mmogs-iii_18.html, 2006.
- [16] Cristalab.com.
Fundamentos básicos del diseño web.
<http://www.cristalab.com/tutoriales/63/fundamentos-basicos-del-diseno-web>, 2008.
- [17] D. Crockford.
JavaScript: The Good Parts.
O'Reilly, Mayo 2008.
- [18] Desarrolloweb.com.
Curso práctico sobre diseño web.
<http://www.desarrolloweb.com/manuales/47>, 2007.
- [19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides.
Design Patterns: Elements of Reusable Object-Oriented Software.

- Addison Wesley Professional, Noviembre 1994.
- [20] S. Limited.
Referente sobre diseño web actual.
<http://www.webdesignfromscratch.com/current-style.cfm>, 2009.
- [21] S. Microsystems.
Java ee tutorial.
<http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html>, 2007.
- [22] Oracle.
Manual para crear *JSP Tag Files*.
http://www.oracle.com/technology/pub/articles/cioroianu_tagfiles.html, 2007.
- [23] F. B. Permuy, M. Álvarez Díaz, and J. R. Santiago.
Apuntes de la asignatura *Integración de Sistemas*.
<http://www.tic.udc.es/~fbellas/teaching/is-2007-2008/index.html>, 2007-2008.
- [24] M. Saltzman.
Como Diseñar Videojuegos. Los Secretos De Los Expertos.
Norma Editorial, 2001.
- [25] Smashingmagazine.
Excelente artículo sobre el diseño de portales web orientado a videojuegos.
<http://www.smashingmagazine.com/2008/08/21/game-sites-design-survey-examples-and-current-practices>,
2008.
- [26] Smashingmagazine.
Importante revista *online* con artículos y ejemplos sobre diseño web.
<http://www.smashingmagazine.com>, 2009.
- [27] W3schools.
Referencia de las propiedades de css2.
http://www.w3schools.com/css/css_reference.asp, 2005.

- [28] Wikipedia.
Ajax (asynchronous javascript and xml).
<http://es.wikipedia.org/wiki/AJAX>.
 - [29] Wikipedia.
Diseño web.
http://es.wikipedia.org/wiki/Diseo_web.
 - [30] Wikipedia.
Internacionalización de aplicaciones.
<http://es.wikipedia.org/wiki/Internacionalizacin>.
 - [31] Wikipedia.
Javaserver pages standard tag library.
http://es.wikipedia.org/wiki/JavaServer_Pages_Standard_Tag_Library.
 - [32] Wikipedia.
Model view controller pattern.
<http://en.wikipedia.org/wiki/Model%-view-controller>.
 - [33] Wikipedia.
Web 2.0.
http://es.wikipedia.org/wiki/Web_2.0.
 - [34] Wikipedia.
Web application *Framework*.
http://en.wikipedia.org/wiki/Web_application_framework.
 - [35] Wikipedia.
Cloud computing.
http://en.wikipedia.org/wiki/Cloud_computing, 2009.
 - [36] Wikipedia.
Colores aplicados al diseño web (paleta websafe).
<http://en.wikipedia.org/wiki/Websafe>, 2009.
-

[37] Wikipedia.

Google app engine.

http://en.wikipedia.org/wiki/Google_App_Engine, 2009.