Universidad de Valladolid

Escuela de Ingeniería Informática de Valladolid

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Mención Computación

Hephaestus: motor de videojuegos en Elixir y su empleo en el estudio de sistemas distribuidos

Autor:
Alonso Sayalero Blázquez

Tutor:
Dr. César Llamas Bello

Índice general

Índice de figuras

Índice de tablas

Índice de Códigos

Capítulo 1

Introducción

Todos los alumnos que cursan alguna carrera universitaria o de grado superior se encuentran con una gran diversidad de métodos de aprendizaje. Durante esos años, los alumnos pueden enfrentar pruebas escritas, orales o prácticas de diversos estilos, algunos más antiguos, otros más modernos.

La docencia, al igual que otras muchas áreas, está en constante evolución, adaptándose a las necesidades del alumnado según las épocas y la tecnología avanzan. Una de las últimas incorporaciones al mundo docente es la gamificación de los conceptos. Esta trata de emplear juegos, diseñados para el caso concreto, con el fin de que el alumno interiorice el concepto que se trata de explicar de una forma más sencilla.

A raíz de esta gamificación surge el problema de cómo implementarla en el ámbito educativo. Este proyecto trata de resolver ese problema para la asignatura de Sistemas Distribuidos de la Escuela de Ingeniería Informática de Valladolid.

Los sistemas distribuidos adquieren más importancia en un mundo en el que todo está conectado por la web y el usuario cada vez tiene menos tolerancia a la ralentización de los servicios a los que accede. Esto obliga a diseñar sistemas distribuidos que permitan un escalado horizontal sencillo y rápido, asegurando la disponibilidad y ajustando el coste a las necesidades puntuales de los usuarios.

Con todo esto en cuenta, la premisa del proyecto es crear un motor de videojuegos que permita arquitecturas tanto de sistemas distribuidos como de no distribuidos. Alrededor del motor, nombrado como **Hephaestus**, se crean un juego, diseñado para el aprendizaje de conceptos de los sistemas distribuidos, haciendo uso del motor y una serie de prácticas para la asignatura, basadas en lo que permite el videojuego.

1.1. Motivación

La motivación aparece a partir de los retos que plantea el proyecto. Dichos retos incluyen el diseño de un motor de videojuegos distribuidos, el diseño de un videojuego pensado para el aprendizaje de un alumno o el empleo de lenguajes de programación que siguen el paradigma funcional.

Todos estos retos suponen un aprendizaje que igual no se daba en otra ocasión y, también, obligan a cambiar el pensamiento imperativo a la hora de programar por uno más flexible que permita la convivencia de ambos paradigmas al mismo tiempo, ya que así lo requieren los lenguajes empleados durante el proyecto.

1.2. Objetivos

El proyecto consta de varios objetivos. El primero se trata de diseñar y programar un motor de videojuegos con las siguientes restricciones:

- Se debe poder utilizar para crear videojuegos tanto distribuidos cómo no distribuidos.
- Debe estar programado en Elixir.
- Debe estar especializado en juegos de tipo rol.

Una vez se cuenta con el motor, el siguiente objetivo es emplear dicho motor y crear un videojuego. El videojuego, al igual que el motor, cuenta con una serie de características que se deben cumplir:

- Se debe diseñar con el fin de enseñar conceptos de los sistemas distribuidos.
- Debido a la restricción del motor, debe estar escrito en Elixir.
- La longitud y complejidad del mismo tiene que estar adaptado al contexto de los escenarios que se le plantean a los alumnos.

Como último objetivo del proyecto se debe proporcionar una serie de prácticas, ya mencionadas en el último punto del listado de características anterior, mediante las cuales el alumno pueda adquirir conocimientos nuevos sobre los sistemas distribuidos y pueda reforzar o comprender conceptos vistos en la teoría de la asignatura. Por ello, estas prácticas o escenarios cuentan también con una serie de restricciones:

- El número de prácticas no puede exceder las tres, debido a que estas forman parte de un conjunto más grande de prácticas que se realizan durante el curso.

- Se debe adaptar la dificultad de las mismas al nivel de los alumnos del curso.
- Se debe adaptar la longitud de las mismas para que, en conjunto con la dificultad, no supongan una inversión de tiempo requerido por el alumno superior a lo establecido por la guía docente.
- Deben estar diseñadas para poder resolverse mediante Java, aunque se puede incluir partes a resolver en Elixir.

1.3. Arquitectura de alto nivel

Se muestra la arquitectura final del proyecto. Esta arquitectura es de alto nivel y por lo tanto no se centra en los detalles, estos se pueden ver en los capítulos 4 y 5.

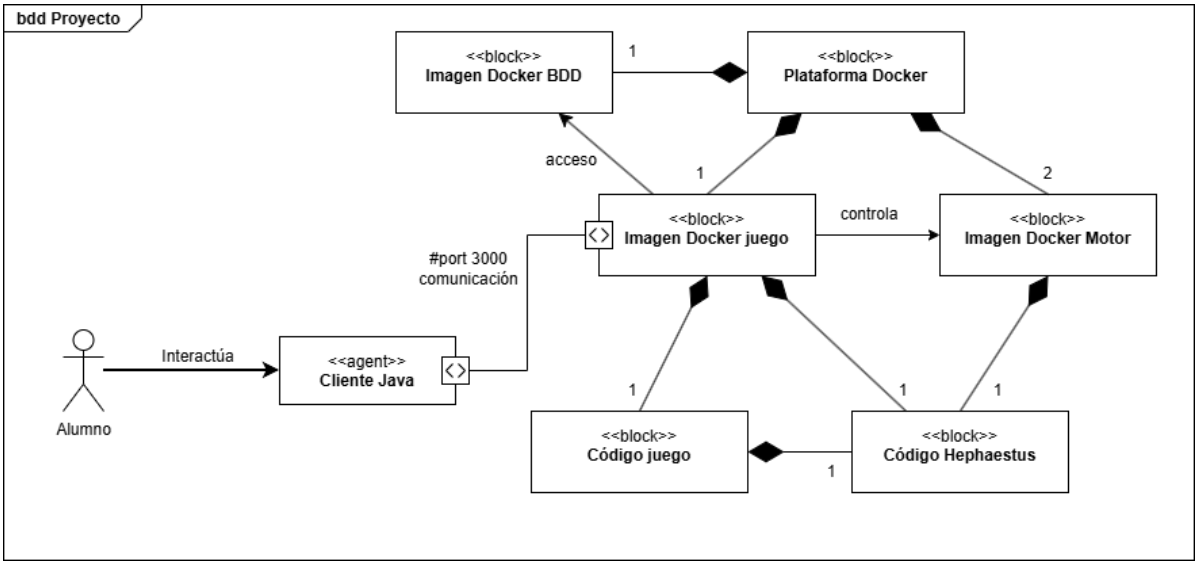

Figura 1.1: Diagrama de bloques de la arquitectura del proyecto

La distribución del código del proyecto se hace mediante Docker, esto permite la simulación de sistemas distribuidos en una única máquina y la posibilidad de desplegar la misma imagen en un sistema verdaderamente distribuido, es decir, en varias máquinas distintas. Todo ello sin necesidad de modificar ningún aspecto de la imagen Docker.

Tal y como se muestra en la figura 1.1, el alumno interactúa, mediante un cliente Java, con la imagen Docker del videojuego mediante el puerto 3000. Dicha imagen contiene el código Elixir del juego y del motor Hephaestus. Esta imagen forma parte de la plataforma Docker encargada de la organización de los contenedores, en los cuales se ejecuta el código de las imágenes. Además de la imagen del juego, la plataforma Docker cuenta con dos imágenes de Hephaestus, las cuales son controladas por la imagen del juego y contienen, únicamente, el código del motor Hephaestus. A la plataforma también se le añade una base de datos Postgresql a la cual se puede acceder desde el juego.

De esta forma se da una visión total de la arquitectura de despliegue final del proyecto. Una plataforma Docker pensada para ser ejecutada en las máquinas de los alumnos, con todo lo necesario para el funcionamiento del proyecto incluido.

1.4. Esta memoria

El documento presentado trata todos los aspectos relacionados con el proyecto que pudieran ser de relevancia.

Siguiendo el orden mostrado en el índice de este documento, los capítulos que se pueden leer son los siguientes: Introducción, planificación, estado del arte, análisis, diseño, pruebas y conclusión.

1.4.1. Introducción

Capítulo en el que se encuentra esta sección, introducción al proyecto y objetivos que se pretenden cumplir con la realización del mismo.

1.4.2. Planificación

En este capítulo se describe la metodología empleada durante la ejecución del proyecto, los artefactos producidos por la metodología empleada y cómo se ha organizado el trabajo a lo largo del tiempo empleado.

También se realiza el análisis de riesgos del proyecto y el análisis de costes, tanto aquellos a la hora de ejecutar el proyecto, como los recurrentes por ejecución a lo largo del tiempo.

1.4.3. Estado del arte

Capítulo sobre otros proyectos con índole similar que hayan podido servir de inspiración. Se describen los proyecto referenciados de manera general y a partir de los aportes realizados a este proyecto.

También se describen las tecnologías empleadas en el proyecto, las características de las mismas y los motivos para su uso.

1.4.4. Análisis

Capítulo centrado en el análisis como ingeniero informático del proyecto. Se incluye un listado de los requisitos funcionales y no funcionales del proyecto, centrado en la pieza principal del mismo, el motor **Hephaestus**. También incluye una descripción escrita y gráfica de la arquitectura del motor y del sistema completo.

1.4.5. Diseño

Este capítulo se centra en el diseño software del sistema, incluyendo la descripción escrita y gráfica del diseño final del motor.

También se incluye en el capítulo, una descripción del diseño del juego y las decisiones tomadas para dicho diseño, así como, una descripción de las prácticas diseñadas para reforzar conocimientos de la asignatura Sistemas Distribuidos.

1.4.6. Pruebas

Capítulo dedicado a las pruebas de calidad realizadas sobre todo el proyecto, centrando la mayor parte sobre el motor.

1.4.7. Conclusión

Capítulo final, últimas reflexiones sobre el proyecto y su ejecución, puntos de enfoque para el desarrollo futuro del mismo y problemas encontrados durante la implementación del código.

Capítulo 2

Planificación

Todo proyecto requiere de una planificación previa que estime el tiempo que conlleva la ejecución del mismo y el coste, tanto del desarrollo del proyecto como de la ejecución a lo largo del tiempo de vida que tenga el resultado final.

A la planificación temporal se le tiene que añadir la elección de tecnologías base con las que se ejecuta el proyecto. Es raro que una tecnología encaje perfectamente en los requisitos para realizar el proyecto, por tanto, estas tecnologías tienen que ser escogidas de acuerdo a un balance entre aspectos positivos que pueden aportar al proyecto y aspectos negativos que puedan suponer un retraso en el mismo.

2.1. Metodología empleada

En el desarrollo software existen múltiples metodologías pensadas para organizar un proyecto, las más empleadas actualmente son el desarrollo en cascada, pionero en la planificación de proyectos de software; Scrum o desarrollo en espiral.

Todas estas metodologías están diseñadas para proyectos en los que existe un producto final claro, que todas las funciones que tiene que realizar ya se han pensado y no van a sufrir modificaciones.

Este proyecto, por el contrario, no cuenta con un estado final claro, evoluciona a lo largo del tiempo. Teniendo esto en cuenta, las metodologías mencionadas anteriormente no pueden ser aplicadas.

Para la situación del proyecto, lo más adecuado es emplear una metodología ágil basada en prototipos. La metodología ágil nace a partir del manifiesto ágil, publicado en 2001, con la intención de cambiar la forma de hacer software [1]. De esta forma el desarrollo de software estaría centrado, según el mencionado manifiesto, en:

- Individuos e interacciones sobre los procesos y herramientas.
- Software funcional sobre extensa documentación.
- Colaboración con el cliente sobre negociaciones contractuales.
- Responder a los cambios sobre seguir un plan rígido.

A partir de entonces múltiples metodologías han surgido siguiendo los principios de este manifiesto y poniéndolos en práctica de diferentes maneras.

Debido a las restricciones de personal y tiempo para el proyecto, la metodología empleada sigue algunos de los principios de las metodologías ágiles *Atern* y *Extreme Programming* tal y como se explican en el libro *Software project managemet* [2]. Los principios que se siguen en el proyecto son los siguientes:

- Entregas a tiempo. Se aplican ventanas de tiempo para realizar las entregas. Cada entrega cuenta con entregables con algún valor propio sin el resto del conjunto del proyecto, evitando el retraso de las fechas de entrega debido a que alguno de los productos sufre algún retraso.
- Desarrollo iterativo. Se crean prototipos los cuales van evolucionando en cada iteración; añadiendo, modificando o eliminando características del entregable; adquiriendo conocimiento sobre el sistema y el proyecto. De esta manera el proyecto se puede modificar añadiendo características que surgen de nuevas ideas derivadas del conocimiento proporcionado por los prototipos. Este punto es fundamental para el proyecto debido a la incertidumbre inicial sobre el producto final.
- Desarrollo incremental. El proyecto se divide en partes más pequeñas las cuales se implementan y entregan de manera secuencial. De esta forma añadir un nuevo componente al proyecto se simplifica al reducirse en añadir una nueva parte.
- Comunicación y retroalimentación. La comunicación se realizará cara a cara siempre que sea posible, haciendo uso de la comunicación telemática en caso de no poderse dar lo anterior. Al estar compuesto el equipo del proyecto por dos personas, se realizará una reunión semanal, siempre que sea posible, en la que se pondrá en común con el tutor los prototipos que el alumno a implementado en ese tiempo. En esa misma reunión se pondrá en común nuevas ideas para la consecución del proyecto y el tutor decidirá con el alumno la siguiente parte del proyecto a comenzar, en caso de haber terminado con la actual.
- Diseño simple. Se prefiere el diseño más simple que cumpla con los requisitos del cliente o del proyecto.
- Refactorización. Se evitará el código espagueti haciendo uso de la refactorización cuando sea necesario. Nunca en partes pequeñas del código y siempre sin miedo a eliminar grandes bloques del mismo si así se requiere.

Con estos principios se pretende dar forma a la idea inicial planteada para el proyecto, añadiendo nuevas ideas y partes durante la creación de las anteriores, hasta llegar al resultado final que se presenta mediante este documento.

2.2. Entregables

Tal y como se ha explicado en el apartado anterior, la metodología empleada para el desarrollo del proyecto hace uso de prototipos iterativos para desarrollar cada una de las partes que conforman el conjunto completo del proyecto, las cuales forman parte del desarrollo incremental de la metodología. Por lo tanto, se tiene que hacer una distinción entre los entregables que pertenecen a la parte incremental de al metodología y aquellos que pertenecen a la parte iterativa. A partir de este punto, en el documento se referirá a aquellos elementos que pertenecen a la parte incremental mencionada como entregables y a aquellos que pertenecen a la parte iterativa como prototipos.

El proyecto cuenta con tres entregables finales: El motor Hephaestus, un juego educativo y un conjunto de prácticas para alumnos de segundo curso del Grado de Ingeniería Informática de la Universidad de Valladolid, cursando la asignatura de Sistemas Distribuidos.

2.2.1. Hephaestus

El motor de videojuegos Hephaestus, cuyas características a alto nivel se han detallado en la sección 1.2, es la pieza central del proyecto, por lo tanto, es lógico que sea el primer entregable de dicho proyecto. El motor final se trata de un conjunto de piezas ya configuradas para su uso en un videojuego, a la vez que contiene utilidades básicas para el manejo del estado del juego, elección de turnos, supervisión de procesos o enrutamiento de procesos distribuidos.

Primer prototipo

Se toma como primera aproximación al motor un prototipo mínimo implementando una pieza mínima del motor, mediante actores, con algunas funciones simples encargadas de modificar el estado interno del actor. Además, se emplea la supervisión de procesos propia de Elixir para el control de procesos vivos como son los actores.

Mediante este prototipo se adquiere conocimiento sobre la implementación de actores en Elixir empleando la biblioteca estándar *GenServer*. También, se obtiene conocimiento sobre el empleo de supervisores de las bibliotecas estándar *Supervisor* y *DynamicSupervisor*.

Segundo prototipo

La evolución lógica al primer prototipo es ampliar la pieza anterior para finalizar su implementación y obtener la pieza final. Esta primera pieza implementa un elemento básico en cualquier videojuego como es el personaje.

A partir de este prototipo se finaliza la implementación de la pieza personaje del motor. Esta pieza está lista para ser utilizada por un juego ya que contiene todas las funciones necesarias para la modificación de su estado interno, así como para devolver información sobre dicho estado.

Tercer prototipo

Una vez se ha adquirido el conocimiento necesario para la implementación de una de las piezas que proporciona Hephaestus al juego, la implementación del resto de piezas se convierte en algo trivial. Siguiendo esta premisa el tercer prototipo está formado por esto mismo, el resto de piezas que Hephaestus proporciona.

El prototipo finaliza con el siguiente conjunto de piezas: Personaje, conexión, enemigo, ubicación, NPC y objeto. Además se adquiere conocimiento sobre la comunicación entre actores.

Cuarto prototipo

Una vez finalizadas las piezas que proporciona Hephaestus, el motor también tiene que incluir ayudas para la gestión del estado del juego, esto es, almacenar permanentemente el estado global de la partida. Para ello se hace uso de una base de datos relacional. Este prototipo es lo mínimo necesario para interactuar desde Elixir con una base de datos.

Se adquiere el conocimiento necesario para conectarse con una base de datos Postgresql, así como, insertar, leer y modificar datos de la misma. En este prototipo todavía no se puede almacenar los datos del juego.

Quinto prototipo

A partir del prototipo anterior, el paso lógico es almacenar el estado del juego en la base de datos y leerlo de vuelta cuando sea necesario.

Se crean las funciones necesarias para este cometido. El resultado permite guardar el estado de todos los actores activos en el momento que se utiliza la función e iniciar los actores almacenados con el estado

guardado. Incluye funciones auxiliares para leer únicamente un tipo de pieza, en caso de ser relevante para el juego construido.

Sexto prototipo

Una función fundamental relacionada con la base de datos es el manejo de los usuarios, la cual no es tratada en el prototipo anterior. En este el objetivo es, precisamente, permitir la creación de nuevos usuarios, con sus respectivos personajes, y, una vez estén creado, que puedan identificarse y obtener su personaje. También se tiene que modificar la función de guardado del juego para que modifique el estado del personaje del jugador que ha guardado la partida.

Al finalizar el prototipo, un juego creado con este prototipo tiene a su disposición todo lo necesario para crear un juego, ya que cuenta tanto con las piezas, las cuales están preparadas para ser empleadas de forma distribuida, para crear el juego como con las utilidades necesarias para guardar el estado de la partida.

Séptimo prototipo

Prototipo final del motor, dando por finalizado este entregable. Se añaden funciones auxiliares para facilitar el desarrollo de juegos de rol distribuidos, evitando al desarrollador la implementación de estas funciones.

El entregable final, en comparación con el prototipo anterior, cuenta con utilidades para seleccionar turnos entre un personaje y un enemigo dependiendo de sus niveles y utilidades para encontrar rutas a partir del identificador de un *GenServer* en una tabla de rutas, la cual contiene pares identificador-dirección del servidor en el que se encuentra el actor.

2.2.2. Juego educativo

El juego educativo es el siguiente entregable después del motor, ya que hace uso de él. Se trata de un videojuego sencillo con dos partes distribuidas, los personajes jugables y los enemigos, de manera que se puede usar para enseñar el funcionamiento de estos sistemas. Cuenta con cuatro habitaciones por las que tiene que pasar el personaje, sin ser opcional ninguna, y presentan una oportunidad de enseñanza sobre algún concepto de los sistemas distribuidos y la programación de actores.

La interfaz de juego está basada en un servidor TCP y comunicación mediante JSON-RPC (3.2) y, al igual que el motor, pasa por varios prototipos para obtener el entregable final.

Primer prototipo

Para comenzar a implementar un juego se tiene que diseñar en primer lugar, este diseño se puede ver con más detalle en el 5. Del entregable anterior se tiene el formato a seguir para que el motor pueda leer de la base de datos el juego.

Se obtiene una traducción del diseño del videojuego a JSON siguiendo el esquema que lee el motor.

Segundo prototipo

La base de cualquier juego es poder jugarlo, para que esto sea posible en este es necesario el empleo de un servidor TCP. Por ello, el primer prototipo es un simple servidor TCP.

El resultado del prototipo es una variación del servidor TCP que presenta la propia documentación de Elixir [3]. Dicha variación permite al servidor admitir diferentes rutas y no solamente actuar como un servidor eco, tal y como lo presenta el ejemplo.

Tercer Prototipo

Una vez existe un servidor TCP que admite diferentes rutas, se tienen que preparar dichas rutas para admitir únicamente JSON-RPC. Este protocolo sigue un formato estricto y cualquier petición que llegue la cual no lo siga tiene que ser rechazada.

De este prototipo se obtiene un servidor TCP funcional y concurrente, el cual solo admite peticiones que sigan el protocolo JSON-RPC con algunas rutas sin funcionalidad real del juego.

Cuarto Prototipo

Con todo el conocimiento adquirido mediante los prototipos anteriores se está preparado para implementar las rutas en el servidor necesarias para poder superar el juego que se ha diseñado.

Este prototipo permite por primera vez jugar al juego de manera no distribuida.

Quinto prototipo

Uno de los requisitos del videojuego es que este tenga partes distribuidas. Para ello se necesitan controladores para cada una de las piezas que se tienen que distribuir en diferentes servidores, en este caso se necesitan controladores para los personajes y los enemigos.

Llegado este punto el juego está ya disponible para poder ser jugado en ambas modalidades, tanto distribuida como no distribuida. Pero, aún así, el entregable no es final.

Sexto prototipo

El servidor TCP creado en el primer prototipo sirve como forma de aprendizaje, pero, como el propio equipo de desarrollo de Elixir explica, no está pensado para producción. Por lo tanto se requiere de una refactorización de la parte del servidor y las rutas del mismo para prepararlo de cara a producción.

La biblioteca que se selecciona para la refactorización es *Thousand Island* [4]. La refactorización del código adaptándolo a esta nueva biblioteca permite la eliminación completa del código del servidor, ya que lo implementa la propia biblioteca. También resulta en una simplificación de las rutas del servidor, disminuyendo la cantidad de código necesaria para hacer lo mismo.

Tras esta refactorización, el juego se da por terminado, siendo este prototipo la versión final del entregable.

2.2.3. Prácticas para alumnos

El siguiente entregable del proyecto consiste en tres prácticas pensadas para reforzar conocimientos obtenidos en segundo curso del Grado de Ingeniería Informática de la Universidad de Valladolid, concretamente, conocimientos de la asignatura de Sistemas Distribuidos. Además de las prácticas el entregable consta de toda la documentación necesaria para que el alumno pueda poner en funcionamiento el juego y de las soluciones a estas.

Primer prototipo

Se inicia la secuencia de iteración creando la primera práctica, esta propone al alumno una introducción a la arquitectura de distribución cliente-servidor y al protocolo JSON-RPC.

Se trata de una primera toma de contacto con un documento didáctico para futuros desarrolladores. Por lo tanto el resultado pone en contexto la idea por la que se mueve la práctica sin llegar a producir un documento didáctico completo.

Segundo prototipo

Una vez la idea de la primera práctica está planteada, se tiene que modificar el documento anterior para que sea adecuado para el usuario objetivo.

Se produce un documento satisfactorio para el nivel del alumno, obteniendo a su vez conocimiento sobre la estructura de un documento de estas características.

Tercer prototipo

Con el conocimiento anterior, se plantea en este prototipo las otras dos prácticas. La segunda práctica del conjunto de tres trata del empleo de actores en los sistemas distribuidos y de las colas de mensajes, elemento fundamental en el funcionamiento de los actores. La tercera y última práctica se adentra en la programación de actores haciendo uso de Hephaestus y permitiendo al alumno modificarlo.

Se crean dos documentos didácticos empleando el conocimiento adquirido con los dos prototipos anteriores. Se finaliza de esta forma todos los documentos referentes a las tres prácticas, pero no es el final del entregable, puesto que faltan las soluciones y el documento de puesta en marcha.

Cuarto prototipo

Es importante tener una referencia de cómo se resuelven las prácticas, de esta forma también se asegura que estas se puedan resolver y no sean imposibles. Debido a que el lenguaje de estudio manejado por la institución es Java, todo código no referente a Hephaestus o el juego se escribe en este lenguaje de programación.

Como resultado se crean tres archivos Java con la solución a cada práctica, otros dos archivos Java con ejemplos para ilustrar algunos conceptos presentados en dichas prácticas y un archivo, también Java, con utilidades para iniciar el juego y sesión con alguno de los personajes.

Quinto prototipo

El último prototipo del entregable es otro documento escrito que sirve como introducción a la sección de las prácticas de la asignatura en la cuál este proyecto se emplea.

Siguiendo el formato de los tres documentos didácticos anteriores, se crea la práctica cero en la cuál se explica lo que se va a hacer a lo largo de las tres prácticas siguientes, se hace una descripción del videojuego, qué tecnologías se van a emplear durante el desarrollo de dichas prácticas y cómo se pone en marcha todo el proyecto.

Mediante este prototipo se da por finalizado este tercer entregable, cumpliendo así todos los requisitos del proyecto.

2.2.4. Distribución

Tal y como se explicó en la metodología empleada, esta nos permite añadir nuevos entregables al final de la línea temporal de ser necesario. En este caso, tras la finalización del proyecto surge la duda de la distribución del mismo. Esto se materializa como un nuevo entregable.

Mediante este entregable se resuelve el problema de simular un sistema distribuido dentro de la máquina empleada por el alumno, así como la distribución del motor y el juego a los mismo alumnos o a cualquier usuario que así lo requiera. Para ello se hace uso de la tecnología Docker.

Primer prototipo

Se crea un archivo imagen de Docker de manera que el juego se pueda ejecutar haciendo uso de esa imagen.

Como resultado, se crea el archivo Dockerfile necesario para la construcción de la imagen del juego. Esta imagen contiene tanto el juego como el motor, ya que es una dependencia del propio juego.

Segundo prototipo

Mediante la imagen creada anteriormente no es posible emplear una arquitectura distribuida, ya que usar múltiples veces la imagen daría lugar a tener varias copias del videojuego incomunicadas entre sí, no a tener un videojuego distribuido. Teniendo en cuenta que el juego se dedica a organizar las piezas que le proporciona el motor, lo que se necesita para distribuir el videojuego es una imagen del motor Hephaestus.

Gracias al conocimiento obtenido en el prototipo anterior, la creación de una imagen Docker del motor es una tarea trivial.

Tercer prototipo

Con las dos imágenes Docker ya creadas, se finaliza el entregable creando un archivo *docker-compose* el cual emplee dichas imágenes para simular un sistema distribuido dentro de la misma máquina.

El resultado se trata de un fichero el cual al ejecutarse mediante Docker levanta dos imágenes del motor y otra más del juego. Además se incluye también la imagen de la base de datos, para hacer más portable el proyecto.

Cuarto prototipo

Al tratarse de un juego distribuido es importante definir las rutas dentro del juego, para que este pueda iniciar cada pieza en la máquina correcta.

Se modifica el código del juego de manera que se emplean rutas hacia la máquina encargada de los personajes para los personajes y, hacia la máquina encargada de los enemigos para los enemigos.

Con esta refactorización del código del videojuego se da por finalizado este entregable.

2.2.5. Web de documentación para desarrolladores de videojuegos

El último entregable del proyecto se trata de una expansión hacia desarrolladores externos, en este caso no para mejorar el motor sino para desarrollar videojuegos empleando el motor. Con ese objetivo se crea una página web empleando la biblioteca *docsify* [5] escrita en Javascript. Esta biblioteca permite escribir documentos Markdown y servirlos como páginas estáticas web.

Primer prototipo

Se prueba el funcionamiento de la navegación haciendo uso de la biblioteca. Esto es especialmente importante debido a que la biblioteca hace uso de un archivo especial en el cuál se tienen que describir las rutas a los archivos creados.

Resulta en una página web inicial con varios documentos Markdown como prueba para navegar entre ellos. Se observa que es más sencillo agrupar cada uno de los grupos de documentación en su propio archivo, aunque este sea largo, y emplear las cabeceras para navegación.

Segundo prototipo

Se crea la página web final, haciendo uso del conocimiento obtenido en el prototipo anterior, con la descripción necesaria de los elementos disponibles en el motor para la creación de videojuegos.

El entregable final cuenta con tres ficheros Markdown en los cuales se describe las piezas proporcionadas por Hephaestus, así como las funciones que estas tienen para poder interactuar con ellas desde Elixir; una descripción sobre cómo está diseñado el motor, con el fin de guiar a los desarrolladores a la hora de crear un juego y una introducción sobre Hephaestus, su origen y motivación.

2.3. Organización temporal

Tal y como se ha mencionado en la sección 2.1, el proyecto está dividido en varios entregables, los cuales a su vez están divididos en varios prototipos. Cada uno de estos prototipos requiere de una retroalimentación con el fin de corregir problemas que puedan haber surgido o, establecer el nuevo camino por el que avanzar una vez el prototipo está finalizado.

En la medida de lo posible, todas las semanas se organiza una reunión entre el tutor y el alumno para que este último ponga en conocimiento común el proceso seguido y el estado del proyecto. En estas reuniones se establece el prototipo que se debe crear para la siguiente reunión, ya sea una corrección del prototipo anterior o el siguiente prototipo necesario para la finalización del entregable actual.

Evento	**Fecha Inicio**	**Fecha Fin**
Reunión inicial sobre el proyecto	13/02/2025	
Investigación inicial	13/02/2025	26/02/2025
Reunión	27/02/2025	
Hephaestus, prototipo 1	27/02/2025	05/03/2025
Reunión	06/03/2025	
Hephaestus, prototipo 2	06/03/2025	12/03/2025
Reunión	13/03/2025	
Hephaestus, prototipo 3	13/03/2025	19/03/2025
Reunión	20/03/2025	
Hephaestus, prototipo 4	20/03/2025	26/03/2025
Reunión	27/03/2025	
Hephaestus, prototipo 5	27/03/2025	02/04/2025
Reunión	03/04/2025	
Hephaestus, prototipo 6	03/04/2025	09/04/2025
Reunión	10/04/2025	
Hephaestus, prototipo 7	10/04/2025	23/04/2025
Reunión	24/04/2025	

Juego, prototipo 1	24/04/2025	07/05/2025
Reunión	08/05/2025	
Juego, prototipo 2	08/05/2025	14/05/2025
Reunión	15/05/2025	
Juego, prototipo 3	15/05/2025	21/05/2025
Reunión	22/05/2025	
Juego, prototipo 4	22/05/2025	28/05/2025
Reunión	29/05/2025	
Juego, prototipo 5	29/05/2025	04/06/2025
Reunión	05/06/2025	
Juego, prototipo 6	05/06/2025	11/06/2025
Reunión	12/06/2025	
Prácticas, prototipo 1	12/06/2025	18/06/2025
Reunión	19/06/2025	
Prácticas, prototipo 2	19/06/2025	25/06/2025
Reunión	26/06/2025	
Prácticas, prototipo 3	26/06/2025	02/07/2025
Reunión	03/07/2025	
Prácticas, prototipo 4	03/07/2025	09/07/2025
Reunión	10/07/2025	
Prácticas, prototipo 5	10/07/2025	16/07/2025
Reunión	17/07/2025	
Parón de verano	21/07/2025	09/09/2025
Reunión	11/09/2025	
Distribución, prototipo 1	11/09/2025	24/09/2025
Reunión	25/09/2025	

Distribución, prototipo 2	25/09/2025	08/10/2025
Reunión	09/10/2025	
Distribución, prototipo 3	09/10/2025	29/10/2025
Reunión	30/10/2025	
Distribución, prototipo 4	30/10/2025	10/11/2025
Reunión	11/11/2025	
Web, prototipo 1	11/11/2025	17/11/2025
Reunión	17/11/2025	
Web, prototipo 2	17/11/2025	24/11/2025
Reunión final	25/11/2025	

Tabla 2.1: Planificación en el tiempo del proyecto

La tabla 2.1 muestra las fechas en las que cada uno de los prototipos ya descritos fueron creados, también incluye las reuniones ya mencionadas y las épocas en las que se produjo un parón.

En algunas ocasiones la posibilidad de hacer una reunión en la semana era nula, debido a sucesos de fuerza mayor, esto se ve reflejado en la tabla mediante saltos de más de una semana entre la fecha inicial y la fecha final de un prototipo.

En total se dedican 31 semanas a la investigación y desarrollo del proyecto, el tiempo medio semanal dedicado es de 16 horas, por lo tanto el tiempo total, en horas, que se dedica al proyecto es una aproximación de 496 horas.

2.4. Riesgos

Un riesgo es una situación potencial que puede acarrear un retraso en el proyecto o lo puede acelerar. Todo riesgo tiene un posibilidad de ocurrir y un impacto sobre el proyecto. Es natural que cualquier proyecto que sea ejecutado esté expuesto a una serie de riesgos que modifiquen los tiempos de ejecución del mismo ya que toda planificación de un proyecto está basada en estimaciones y sujeta a incertidumbres.

Todo proyecto de desarrollo software requiere de un análisis de riesgos que se puedan producir durante la ejecución del mismo. Los riesgos se dividen en diversos tipos dependiendo del área al que afecten:

- Estimación. Fallos a la hora de estimar los tiempos de desarrollo o el tamaño del software, entre otros muchos casos.
- Organizativos. Reestructuraciones en mitad del proceso de desarrollo, problemas económicos o derivados de estos.
- Personas. Falta de personal, enfermedades o personal sin el entrenamiento adecuado son algunos de los riesgos que pertenecen a esta categoría.
- Requisitos. Principalmente cambios de requisitos que supongan grandes cambios en el diseño.
- Tecnológicos. Bases de datos que no consiguen procesar todas las transacciones que se esperan, por ejemplo.
- Herramientas. Fallos de hardware o de herramientas de generación de código forman parte de este tipo de riesgos.

El análisis de riesgos realizado cuenta con los siguientes elementos, los cuales identifican, en la medida de lo posible, todas las variables para la prevención y recuperación de los riesgos [2] [6].

- **Probabilidad**: Nivel de posibilidad de que el riesgo suceda. Puede ser baja, media o alta.
- **Impacto**: Nivel del impacto que tiene el daño producido en caso de que el riesgo suceda sobre la ejecución del proyecto. Puede ser muy bajo, bajo, medio, alto o muy alto.
- **Plan de mitigación**: Lista de acciones que se pueden tomar para reducir la probabilidad de que le riesgo suceda.
- **Plan de contingencia**: Lista de acciones que se pueden llevar a cabo una vez el riesgo ha sucedido para reducir el impacto sobre los tiempos de ejecución del proyecto.

Las siguientes tablas describen los riesgos identificados en el análisis de riesgos del proyecto. A parte de los elementos descritos anteriormente, cada tabla incluye también una descripción del riesgo al que se refieren.

Riesgo R01	**Falta de disponibilidad**
Descripción	Durante el desarrollo pueden surgir imprevistos, ya sean de salud o de otra índole, que impidan la participación de alguno de los dos integrantes del proyecto, el impacto será mayor si es el alumno el que sufre un imprevisto que le impida avanzar con el proyecto.
Probabilidad	Media
Impacto	Medio
Plan de mitigación	▪ Evitar situaciones que puedan poner en riesgo la salud o la integridad física.
Plan de contingencia	▪ Aprovechar periodos de parada o con baja carga de trabajo para recuperar el tiempo que se pudiera haber perdido. ▪ Considerar tomar vacaciones en el trabajo para dedicarle más tiempo al proyecto.

Tabla 2.3: Riesgo R01

Riesgo R02	**Fallos de hardware**
Descripción	Todo equipo electrónico puede llegar a fallar, lo que puede suponer retrasos y perdida de información.
Probabilidad	Baja
Impacto	Muy alto
Plan de mitigación	▪ Disponer de un dispositivo de repuesto. ▪ Realizar copias de seguridad de todo lo relacionado con el proyecto de forma habitual. ▪ No mantener las copias de seguridad en el mismo equipo que el que se usa para trabajar en el proyecto, de ser posible almacenarlas en la nube. ▪ Instalar sistemas SAI para evitar perder datos ante un corte de luz.
Plan de contingencia	▪ Recuperar las copias de seguridad, de existir, en otro dispositivo. ▪ En caso de archivos corruptos intentar usar programas de recuperación para este tipo de archivos.

Tabla 2.5: Riesgo R02

Riesgo R03	**Error en la estimación de tiempo de desarrollo o tamaño de software**
Descripción	La novedad para el alumno de muchas de las tecnologías empleadas pueden provocar en la incorrecta estimación tanto del tiempo que puede llevar la ejecución de un prototipo como el tamaño de dicho prototipo. Una mala estimación del tamaño conlleva también un retraso en el proyecto.
Probabilidad	Alta
Impacto	Muy alto
Plan de mitigación	▪ Realizar análisis en profundidad de las tareas para obtener estimaciones más precisas. ▪ Dar prioridad a las partes del proyecto que resulten críticas. ▪ Emplear tiempo extra de tareas sencillas o cuya estimación haya sido positiva para aprender en mayor profundidad las tecnologías empleadas.
Plan de contingencia	▪ Reorganizar el proyecto eliminando partes no importantes de manera que su objetivo principal no se vea afectado. ▪ En última instancia, retrasar el final del proyecto para poder incluir todos los aspectos básicos del mismo.

Tabla 2.7: Riesgo R03

Riesgo R04	**Falta de conocimiento sobre las tecnologías**
Descripción	Aunque se inicia el proyecto sabiendo que una de las tecnologías supone un nuevo reto para el alumno (Elixir), existe el riesgo de subestimar la complejidad de dicha tecnología o sobrestimar el conocimiento sobre el resto de tecnologías que puedan llegar a ser empleadas en el proyecto.
Probabilidad	Media
Impacto	Alto
Plan de mitigación	▪ Estudio de la documentación oficial de las tecnologías empleadas. ▪ Seleccionar tecnologías conocidas para las partes del proyecto en las que no haya que emplear *Elixir*. ▪ Contar con un conjunto de tecnologías que puedan reemplazar alguna de las ya empleadas.
Plan de contingencia	▪ Cambiar tecnologías las cuales supongan una mayor complejidad a la esperada con alguno de los sustitutos investigados. ▪ Redistribuir los tiempos de desarrollo para priorizar las tecnologías que conlleven un mayor tiempo debido al escaso conocimiento. ▪ Emplear herramientas o bibliotecas que simplifiquen las partes menos importantes.

Tabla 2.9: Riesgo R04

Riesgo R05	**Cambios bruscos en los requerimientos que obliguen a un rediseño**
Descripción	Debido a la naturaleza del proyecto, pueden llegar a surgir cambios en los requerimientos de alguno de sus componentes que obliguen a realizar un rediseño completo del componente afectado, suponiendo un retraso en los tiempos del resto de prototipos y entregables
Probabilidad	Baja
Impacto	Medio
Plan de mitigación	▪ Optar por una implementación modular, debido a que un cambio de diseño puede afectar a unos pocos módulos y no a todo código. ▪ Estudiar los puntos del proyecto que puedan ser más propensos a recibir muchos cambios de requisitos y no implementarlos hasta que sea estrictamente necesario. ▪ Hacer uso de herramientas de generación de código que puedan asistir con la modificación del código.
Plan de contingencia	▪ Descartar cambios de requisitos cuyo aporte al proyecto frente al coste que suponga implementarlos sea muy negativo. ▪ Eliminar requisitos menos importantes en favor de cambios a requisitos importantes. ▪ En caso de falta de tiempo, adaptar los requisitos al código y no al revés.

Tabla 2.11: Riesgo R05

Riesgo R06	**Fallo en el uso de herramientas de inteligencia artificial**
Descripción	El avance en las herramientas de inteligencia artificial es claro y muy útil para el desarrollo de software, pero, al igual que otras herramientas, la inteligencia artificial puede fallar y entorpecer el desarrollo en lugar de acelerarlo.
Probabilidad	Baja
Impacto	Bajo
Plan de mitigación	▪ No emplear las herramientas para la generación de código, hacer uso de ellas únicamente para búsqueda de fallos o asistencia con la refactorización del código. ▪ Supervisar el trabajo realizado por la inteligencia artificial. ▪ Si se emplea en la refactorización de grandes bloques de código, implementar la refactorización de manera gradual y probando si el nuevo código no falla.
Plan de contingencia	▪ Si se detectan fallos continuados sin mejora, dejar de hacer uso de las herramientas.

Tabla 2.13: Riesgo R06

Riesgo R07	**Fallos de entendimiento a la hora de definir los requerimientos**
Descripción	Toda comunicación humana es propensa a que se produzcan malentendidos. Una mala comunicación entre tutor y alumno puede suponer retrasos en el proyecto debido a reescrituras de código que, de no haberse producido el malentendido, no habrían sido necesarias.
Probabilidad	Baja
Impacto	Muy bajo
Plan de mitigación	▪ Asegurar la claridad de la idea antes de proponer requerimientos. ▪ Estudiar todas las posibles ramificaciones que puedan surgir de un requerimiento.
Plan de contingencia	▪ Adaptar un punto medio entre la idea original y el código producido. ▪ Eliminar requerimientos no esenciales para el proyecto en el caso de que produzcan mucha confusión.

Tabla 2.15: Riesgo R07

2.5. Costes

Además del análisis de riesgos de un proyecto, es crucial realizar también un análisis de costes del mismo. Dicho análisis tiene que tener en cuenta los costes acarreados en la etapa de desarrollo del proyecto y los costes que va a tener el producto final durante su vida útil.

Cada una de las etapas cuenta con costes comunes entre sí y costes que son específicos de la etapa concreta. A su vez, los costes de cada etapa se pueden dividir en costes recurrentes o costes fijos. Un coste recurrente es todo aquel coste que se produzca de manera periódica, por ejemplo una subscripción a un servicio, mientras que un coste fijo es aquel que solo se produce en un momento concreto y, una vez realizado el desembolso, no hay necesidad de que se vuelva a producir, por ejemplo la compra de un equipo.

2.5.1. Desarrollo

Durante el desarrollo del proyecto, se producen unos costes relacionados con la mano de obra, el equipo empleado, el coste de alojamiento del control de versiones del código, las licencias de software necesarias y el alojamiento web.

Mano de obra

El gobierno de España establece el salario mínimo mensual en 1184 euros en 14 pagas [7]. Si se tiene en cuenta que semanalmente se le han dedicado 16 horas al proyecto de media, se puede aproximar el sueldo mensual a 518 euros [8]. Habiéndose alargado el desarrollo del proyecto hasta los 10 meses, el coste aproximado de mano de obra es de 5180 euros.

Equipo empleado

Para el desarrollo del proyecto se han empleado dos equipos, un ordenador fijo de sobremesa, para el desarrollo diario del mismo, y un ordenador portátil, para aquellas ocasiones en las que se necesitara la portabilidad que ofrece un equipo de estas características.

El coste del ordenador de sobremesa es de 1600 euros mientras que el portátil se valora en 700 euros. Estos costes se toman del precio pagado en el momento de la compra de los mismo.

En total, el coste en equipo de trabajo es de 2300 euros.

Control de versiones

El código se aloja en un servicio de control de versiones en la nube llamado Github. El modelo de precios de Github no supone coste alguno para todo aquel repositorio de código que sea público, tal y cómo es este proyecto. Por lo tanto el coste de alojamiento del control de versiones es de 0 euros.

Licencias de software

El uso del lenguaje de programación Elixir es gratuito, al igual que el empleo de Docker y Postgresql.

Por otro lado, Java tiene un modelo de financiación variado en el cual el empleo del lenguaje de programación en su variante empresarial es de pago, mientras que la propia Oracle mantiene una implementación de la plataforma Java, licenciada baja la licencia GPL-2.0, llamada Open-JDK. Esta variante sufre actualizaciones cada semestre por lo que normalmente se encuentra por detrás de la versión empresarial. Para el empleo dado en el proyecto, la versión libre y gratuita es suficiente.

El editor empleado para el desarrollo del proyecto es Visual Studio Code, editor de texto creado por Microsoft y que está orientado a la programación, supone un coste nulo, ya que es gratuito.

Por último, se hace uso de ChatGPT como herramienta de inteligencia artificial. Pese a que este cuenta con varias opciones de pago, el nivel básico gratuito es suficiente para las tareas acometidas durante el desarrollo del proyecto.

Con todo esto en cuenta, el coste de licencias de software es de 0 euros.

Alojamiento web

Una de las partes del proyecto es una página web pensada para todo aquel desarrollador de videojuegos que quiera hacer uso del motor Hephaestus. Esta web tiene que ser alojada en un servidor web para que sea visible públicamente. Github ofrece alojamiento de páginas web estáticas sin ningún coste para repositorios de código públicos.

Por lo tanto el coste del alojamiento es también de 0 euros y el coste de desarrollo de todo el proyecto asciende a 7480 euros.

2.5.2. Producción

Una vez finalizado el proyecto la etapa de producción del mismo puede tener unos costes asociados. Al igual que a la etapa de desarrollo, a esta etapa también le afectan los costes de licencias de software,

alojamiento web y alojamiento del control de versiones. Estos costes ya han sido establecidos como nulos en el apartado anterior.

Los costes propios de la etapa de producción son el alojamiento de las imágenes Docker del proyecto y el equipo necesario para ponerlo en funcionamiento.

Alojamiento de imágenes

Las imágenes Docker del proyecto se pueden construir a partir del código disponible en el repositorio de Github, pero, en caso de querer almacenar las imágenes ya construidas en un repositorio público para un acceso más sencillo, existen productos que ofrecen estos servicios.

DockerHub es el repositorio en la nube más conocido. Propiedad también de los dueños de Docker, su modelo de precios permite el alojamiento de una imagen privada, en caso de necesitar más se tiene que optar por una opción de pago, y alojamiento ilimitado de imágenes públicas.

En el caso del proyecto las imágenes son públicas, por lo tanto coste 0.

Hardware

El proyecto completo está pensado para poder ser usado en una sola máquina y, como se estableció en el apartado de desarrollo, un ordenador valorado en 700 euros es capaz de ejecutarlo.

A parte de poder ser usado en una sola máquina, la naturaleza del proyecto es distribuida y por lo tanto se puede desplegar de esa manera. Para ello son necesarias tres máquinas independientes, si se opta por el empleo de VPS (Servidor Virtual Privado, por sus siglas en inglés) el coste se puede estimar en 4 euros mensuales por máquina.

Por lo tanto, si se opta por la primera opción, el coste de la producción del proyecto es únicamente de un mínimo de 700 euros como coste fijo. En caso de optar por la segunda opción el coste sería recurrente costando 12 euros mensuales, dependiendo siempre del distribuidor de VPS elegido.

Capítulo 3

Estado del arte

Aunque una idea sea original existen otros muchos trabajos que pueden servir para aportar nuevas ideas, encontrar soluciones a problemas que pueda llegar a tener el proyecto o simplemente afianzar la idea inicial al ver otro proyecto que comparte un concepto que se va a emplear en el nuevo.

La construcción del proyecto depende enteramente de las tecnologías que lo conforman, sin estas el proyecto no tendría entidad y solamente existiría como concepto abstracto.

3.1. Antecedentes

Dado que el foco principal del proyecto es el motor de videojuegos distribuidos Hephaestus, el foco de la investigación inicial es la búsqueda de otros ejemplos de videojuegos distribuidos y, por lo tanto, que cuenten con un motor distribuido.

3.1.1. Second Life

Second Life es un mundo virtual multijugador que permite a los usuarios crearse un avatar para ellos mismos e interactuar a través del avatar con otros usuarios y con el contenido creado por estos. Fue lanzado en Junio de 2003 por la compañía Linden Lab.

Se puede acceder al mundo virtual mediante el cliente software propio de la empresa o mediante clientes creados por terceros, ya que la tecnología del cliente es de código abierto.

La tecnología sobre la que se construye Second Life está formada por el cliente, ejecutado en la máquina del usuario, y miles de servidores bajo el mando de Linden Lab [9].

Cliente

El cliente renderiza gráficos 3D utilizando la tecnología OpenGL y su código fuente está licenciado bajo LGPL desde 2010, permitiendo la creación de clientes por partes de terceros los cuales añaden funcionalidades que el oficial no tiene.

Servidor

Cada región del mundo virtual (256x256 metros de área) se ejecuta en un núcleo en un servidor con un procesador multinúcleo. Además del alojamiento de la región del mundo correspondiente los servidores se encargan de la comunicación entre usuarios y objetos presentes en la región de la que se encargan. Además realizan los cálculos de colisiones entre avatares y objetos.

Figura 3.1: Algunos avatares dentro del juego. Imagen por HyacintheLuynes

De este proyecto se puede obtener la confirmación del uso de servidores distribuidos encargados cada uno de un elemento concreto. Aunque todos corren el mismo conjunto de operaciones, cada uno se especializa en una de las regiones del mundo virtual y se comunica con el resto de servidores y con los clientes para conservar la integridad de dicho mundo.

Esto se puede extrapolar al proyecto confirmando que el motor puede ser distribuido sin realizar modificaciones al código del mismo y que cada servidor se puede especializar en un elemento concreto del juego.

3.1.2. JominiEngine

El JominiEngine es un motor de videojuegos distribuidos creado por David Alexander Bond en 2015 como parte del estudio *Design and implementation of a massively multi-player online historical role-playing game*. En este estudio se presenta la funcionalidad imprescindible del motor así como guías para su futuro desarrollo [10].

Este proyecto permite confirmar la posibilidad de hacer algo parecido con este proyecto. Pese a que la premisa de motor de videojuegos distribuidos de rol es igual para ambos proyectos, el JominiEngine está especializado en juegos de rol históricos, mientras que Hephaestus se enfoca en la fantasía clásica popularizada por juegos de rol de mesa como Dungeons and Dragons.

Se pueden encontrar otras muchas diferencias entre motores debido a las diferencias entre requisitos de un proyecto al otro, mientras que JominiEngine enfoca su evolución hacia los juegos MMORPG, Hephaestus tiene un enfoque más didáctico y de un jugador, aunque esto no impide que se utilice en juegos multijugador.

Pese a las diferencias JominiEngine es un buen punto de conocimiento para el desarrollo de motores de videojuegos que permitan la creación de juegos distribuidos.

3.2. Tecnologías empleadas

La elección de tecnologías se realiza teniendo en cuenta las necesidades del proyecto. Estas necesidades que se necesitan suplir son los lenguajes de programación en los que se va a realizar el proyecto, el gestor de bases de datos empleado y otras herramientas o tecnologías necesarias para el correcto desarrollo del mismo.

3.2.1. Lenguajes de programación

Un proyecto software está sostenido por los lenguajes de programación en los que se construye. La elección de lenguajes de programación se guía por las características que estos tengan o por requisitos que tenga el proyecto.

Elixir

Elixir [11] es un lenguaje de programación que ofrece unas características que lo hacen especialmente apto para el diseño y programación de sistemas distribuidos. Sigue el paradigma de programación funcional y está diseñado para ser implementado en una arquitectura modular.

Es un lenguaje basado en Erlang y emplea la misma máquina virtual que este. De esta manera se puede hacer uso de bibliotecas originalmente pensadas para Erlang desde el propio Elixir de forma nativa.

Además, Elixir ofrece otra serie de características, algunas de ellas heredadas de Erlang, que lo siguen posicionando como un lenguaje muy interesante para el desarrollo de sistemas distribuidos y desarrollo web:

- **Tolerancia a los fallos**. Si algún componente del sistema falla, es posible recuperar la parte del sistema en la que se encuentra este fallo sin poner en riesgo el funcionamiento del resto del sistema. Esto se debe al sistema de supervisores que Elixir hereda de Erlang.
- **Distribuido**. Elixir puede ejecutar partes del código en procesos ligeros que están aislados y se comunican mediante mensajes. Estos procesos se pueden ejecutar en cualquier máquina que cuente con una máquina virtual de Erlang y el propio lenguaje cuenta con mecanismos nativos para facilitar la comunicación entre procesos localizados en máquinas diferentes, siempre que estas se puedan comunicar entre sí.
- **Escalable**. Gracias a la característica del punto anterior, el código de Elixir se puede lanzar nuevos procesos sin afectar el funcionamiento del resto. Estos nuevos procesos se pueden lanzar en máquinas diferentes si se requiere.

Todas estas características le han hecho posicionarse como un lenguaje muy interesante en el mundo de la programación web, gracias al framework de desarrollo de aplicaciones web Phoenix [12] y lo hacen el lenguaje elegido para la implementación del motor de videojuegos y el juego didáctico.

Java

Java [13] es un lenguaje de programación diseñado para seguir el paradigma de programación orientada a objetos y cuyas fortalezas residen en:

- **Independiente de la plataforma**. Java es un lenguaje de programación compilado. El resultado de los lenguaje compilados, tradicionalmente, está sujeto a la arquitectura del sistema operativo en el que se compila y, por ejemplo, un binario de un lenguaje compilado en Windows no se puede ejecutar en Linux. Java toma otro acercamiento y se ejecuta sobre una máquina virtual, la JVM (Java Virtual Machine). Gracias a esto, el resultado de la compilación es un binario que se puede ejecutar sobre cualquier JVM independientemente del sistema operativo sobre el que se ejecute. El único requisito es que el sistema operativo sobre el que se ejecuta el programa tiene que tener instalada la máquina virtual de Java.

- **Fuertemente tipado**. Los lenguajes de programación fuertemente tipados evitan la asignación de valores a variables cuando el tipo de ambos no coincide y obligan a asignarle un tipo a cada variable, no pudiendo haber variables sin tipo hasta que se las asigna un valor. Esto evita errores durante la programación que puedan llevar al programa a fallar durante la ejecución.
- **Recolector de basura**. Java cuenta con un recolector de basura. El recolector de basura es un mecanismo que tienen algunos lenguajes de programación que permite liberar espacio de memoria de manera automática en tiempo de ejecución. Para ello el recolector de basura libera el espacio ocupado por variables que ya han sido utilizadas y no tienen uso en el futuro de la ejecución.
- **Amplia comunidad**. Es un lenguaje utilizado en multitud de sistemas que aún siguen en funcionamiento y, pese a su antigüedad, sigue siendo escogido por multitud de empresas y personas. Esto hace que Java cuente con una gran comunidad de programadores e ingenieros y por consiguiente se pueden encontrar una gran cantidad de recursos para cualquier necesidad que pueda surgir durante un proyecto.

Estas características hacen de Java un lenguaje muy asequible para los alumnos que se inician en la carrera universitaria de la ingeniería informática. Por este motivo y debido a que es el lenguaje en el que se realizan el resto de prácticas de la asignatura, se emplea Java para proporcionar las soluciones a las prácticas de los alumnos.

SQL

Structured Query Language [14], más conocido como SQL, es el lenguaje de programación de dominio específico diseñado para gestionar datos estructurados en sistemas de gestión de bases de datos o RDBMS por sus siglas en inglés.

SQL permite al motor Hephaestus comunicarse con la base de datos relacional elegida y su uso es obligatorio debido a que dicha base de datos no admite otro lenguaje. Para ello se hace uso de una biblioteca de Elixir llamada Ecto, la cual permite hacer consultas a la base de datos mediante una abstracción propia o mediante código SQL directamente. Esta biblioteca emplea SQL para sus consultas aunque no se haga uso de la opción de código SQL directo.

JSON

JSON (JavaScript Object Notation) [15] es un formato de intercambio de datos diseñado para poder ser legible para los humanos. Basado en un subconjunto de JavaScript [16], es completamente independiente a este y puede ser empleado para transmitir datos entre diferentes lenguajes de programación.

Es el lenguaje empleado por el protocolo admitido por el juego didáctico, el cual se verá en el apartado de otras tecnologías de esta misma sección, y también se usa para almacenar los estados de los actores en la base de datos, por lo tanto es el lenguaje que define el juego.

3.2.2. Gestor de bases de datos

El gestor de bases de datos es el encargado de procesar las consultas realizadas en SQL y modificar la base de datos acorde a esas consultas de ser necesario.

Postgresql

Postgresql [17] es un gestor de bases de datos relacionales de código abierto. Cuenta con su propia extensión del lenguaje SQL y admite tanto la extensión como SQL estándar. Se caracteriza por su robustez, seguridad y rendimiento. Asegura la integridad de los datos al realizar operaciones sobre estos y cuenta con una gran variedad de tipos de datos que pueden ser almacenados. Además cuenta con la posibilidad de extender sus capacidades mediante plugins.

Se elige este gestor debido a sus grandes capacidades y a su extenso catálogo de tipos de datos, el cual incluye JSONB un tipo de dato que permite el almacenamiento de documentos JSON en formato binario y la capacidad de hacer cálculos sobre estos.

3.2.3. Otras tecnologías

Docker

Docker [18] es un sistema de gestión y ejecución de contenedores. Un contenedor es una pieza de software que simula un sistema operativo reducido dentro del sistema operativo de la máquina que ejecuta Docker, permitiendo ejecutar programas dentro de este sistema operativo en miniatura empleando los recursos del sistema operativo donde se ejecuta el contenedor. Para la construcción o ejecución de un contenedor se emplean imágenes, estas imágenes contienen la definición del contenedor y del código que se ejecuta dentro de este. Esto le permite a Docker contar con varias ventajas a la hora de distribuir programas a servidores o usuarios:

- **Control de versiones**. Al crear un contenedor este tiene la capacidad de otorgarle una etiqueta con la versión de la que se trata. Esto permite que el cambio de versiones consista únicamente en crear una nueva imagen con la nueva versión del código y una nueva etiqueta de versión.
- **Portabilidad**. Los contenedores se pueden ejecutar en cualquier sistema con Docker instalado.

- **Escalabilidad**. Existen tecnologías que permiten la duplicación de contenedores, el balance de carga sobre contenedores duplicados y creación y destrucción automática de contenedores. Una de las tecnologías más populares que permite esto es Kubernetes [19].

Aunque la portabilidad de Docker no es de mucha utilidad ya que no proporciona nada nuevo a lo que ya ofrecen Elixir y Java, Docker permite simular un sistema distribuido en la misma máquina física sin necesidad de configurar máquinas virtuales. Esto supone una ventaja a la hora de que los alumnos instalen el proyecto al solo necesitar un archivo *docker-compose* para poner todo el sistema en funcionamiento.

JSON-RPC

JSON-RPC [20] es un protocolo de hilo basado en JSON para realizar llamadas de procedimiento remoto (RPC por sus siglas en inglés). Este protocolo permite notificaciones al servidor y el envío de múltiples llamadas al servidor, las cuales pueden ser respondidas asíncronamente o síncronamente.

El protocolo es independiente del protocolo de transporte empleado, en este proyecto se transporta empleando el protocolo TCP. Otra de las características es que no incluye ningún tipo de autenticación o autorización, se tiene que implementar aparte.

Se trata del protocolo empleado en el proyecto para la comunicación entre los clientes Java y el juego didáctico. Transportado, como ya se ha mencionado, mediante TCP.

Capítulo 4

Análisis

El proyecto cuenta con tres grandes bloques de contenido con código: el motor Hephaestus, el juego didáctico y las prácticas para los alumnos. El código de las prácticas para los alumnos, al tratarse de las soluciones de estas, es sencillo y orientativo, por lo tanto queda excluido del análisis formal del proyecto.

4.1. Motor Hephaestus

Se trata del foco principal del proyecto, por lo tanto el grueso del análisis se concentra en el motor de videojuegos distribuidos Hephaestus.

4.1.1. Requisitos

Hephaestus se describe como un motor para el desarrollo de videojuegos distribuidos escrito en Elixir. Por lo tanto dicho motor deberá contar, fundamentalmente, con: la capacidad de ser distribuido; la posibilidad de emplearlo para manejar el almacenamiento del estado del juego, ya sea de manera permanente o efímera; la capacidad de simplificar el desarrollo de videojuegos y la modularidad suficiente para admitir futuras extensiones de capacidades.

Con esta descripción como base, a continuación se muestran las tablas de requisitos funcionales y no funcionales del motor Hephaestus.

Requisito
Tiene que tener la capacidad de autenticar y autorizar a los usuarios.

Cada pieza que actúe como actor debe tener un identificador único que sirva para su identificación en todo el sistema.
Tiene que permitir la creación de los siguientes actores: Personaje, conexión, enemigo, ubicación, NPC y objeto.
Los actores personaje, NPC y enemigo tienen que tener las siguientes estadísticas: Carisma, sabiduría, inteligencia, constitución, destreza y fuerza.
Todos los actores tienen que poder devolver el estado actual en el que se encuentran.
El actor personaje tiene que poder almacenar objetos en un inventario.
El actor personaje tiene que poder equiparse objetos de su inventario para aumentar o disminuir sus estadísticas.
El actor personaje tiene que poder subir de nivel tras llegar a un umbral de experiencia conseguida.
El actor personaje, el actor NPC y el actor enemigo deben contar con un nivel con el fin de realizar resoluciones de conflictos.
El actor personaje y el actor enemigo deben contar con un contador de vida que al llegar a cero cambie su estado a no vivo pero no destruya el actor.
El actor personaje tiene que indicar si se encuentra en combate o no.
El actor personaje tiene que tener al capacidad de recordar la ubicación en la que se encuentra y cambiarla si se requiere.
Tiene que tener los mecanismos necesarios para asociar un personaje a un usuario.
El actor enemigo tiene que proporcionar una cantidad de experiencia arbitraria si su vida a llegado a cero.
El actor enemigo tiene que contar con una estadística a partir de la cual se derive el daño de ataque que tiene.
El actor NPC tiene que contar con un resultado para una interacción exitosa.
El actor NPC tiene que contar con un límite de interacciones que pueda ser opcional. En caso de no ser opcional, el resultado de obtener la interacción debe ser exitoso hasta que se consuman todos los intentos.
El actor conexión debe almacenar exactamente dos ubicaciones.
El actor conexión debe mostrar la siguiente ubicación en función de la ubicación actual.

El actor conexión debe tener un nivel fijo.
El actor conexión debe tener la capacidad de tener un objeto que se pueda emplear desde el juego para no permitir el cruce hasta la obtención del objeto.
El actor objeto tiene que tener un nivel fijo.
El actor objeto tiene que tener el tipo de objeto del que se trata.
El actor objeto tiene que tener un valor numérico que represente diferentes cosas dependiendo del tipo de objeto.
El actor objeto tiene que tener al opción de añadir una estadística sobre la que actúe el objeto.
El estado del juego se tiene que poder almacenar de forma permanente en una base de datos.
Tiene que proporcionar utilidades para simplificar el enrutamiento de actores en arquitecturas distribuidas.
Tiene que proporcionar utilidades para la supervisión de procesos.

Tabla 4.1: Hephaestus. Requisitos funcionales

Requisito
Tiene que estar escrito en el lenguaje de programación Elixir.
Tiene que estar especializado en juegos de rol en mundos de fantasía.
Las contraseñas deben ser almacenadas empleando una función hash.
Tiene que estar pensado para su futura modificación haciendo uso de una arquitectura de módulos.
La definición de tipos de objetos tiene que ser dependiente del juego que emplea el motor.
La base de datos empleada debe ser Postgresql.
El control de los actores debe residir en el motor.

Tabla 4.2: Hephaestus. Requisitos no funcionales

4.1.2. Modelos

En esta sección se muestran los diagramas del análisis llevado a cabo sobre el motor. En primer lugar el análisis inicial muestra el análisis completo del motor.

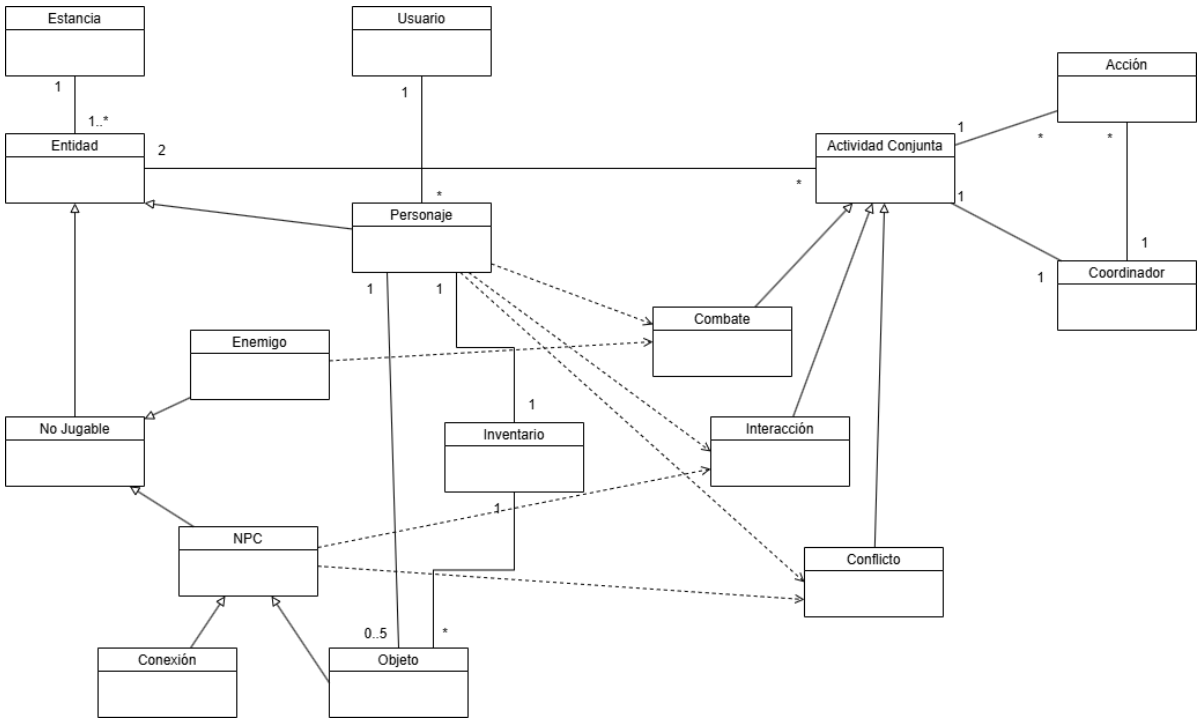

Figura 4.1: Diagrama de análisis Hephaestus completo

De este análisis inicial del motor completo, se tienen que eliminar algunos elementos, que no suponen la imposibilidad de crear juegos sin ellos, debido a la falta de tiempo que supone la implementación completa del motor, además del resto del proyecto. Los elementos eliminados son los siguientes:

- Acción.
- Coordinador.
- Acción conjunta.
- Combate.
- Interacción.
- Conflicto.

Una vez eliminados estos componentes el diagrama final de análisis del motor Hephaestus queda de la siguiente forma.

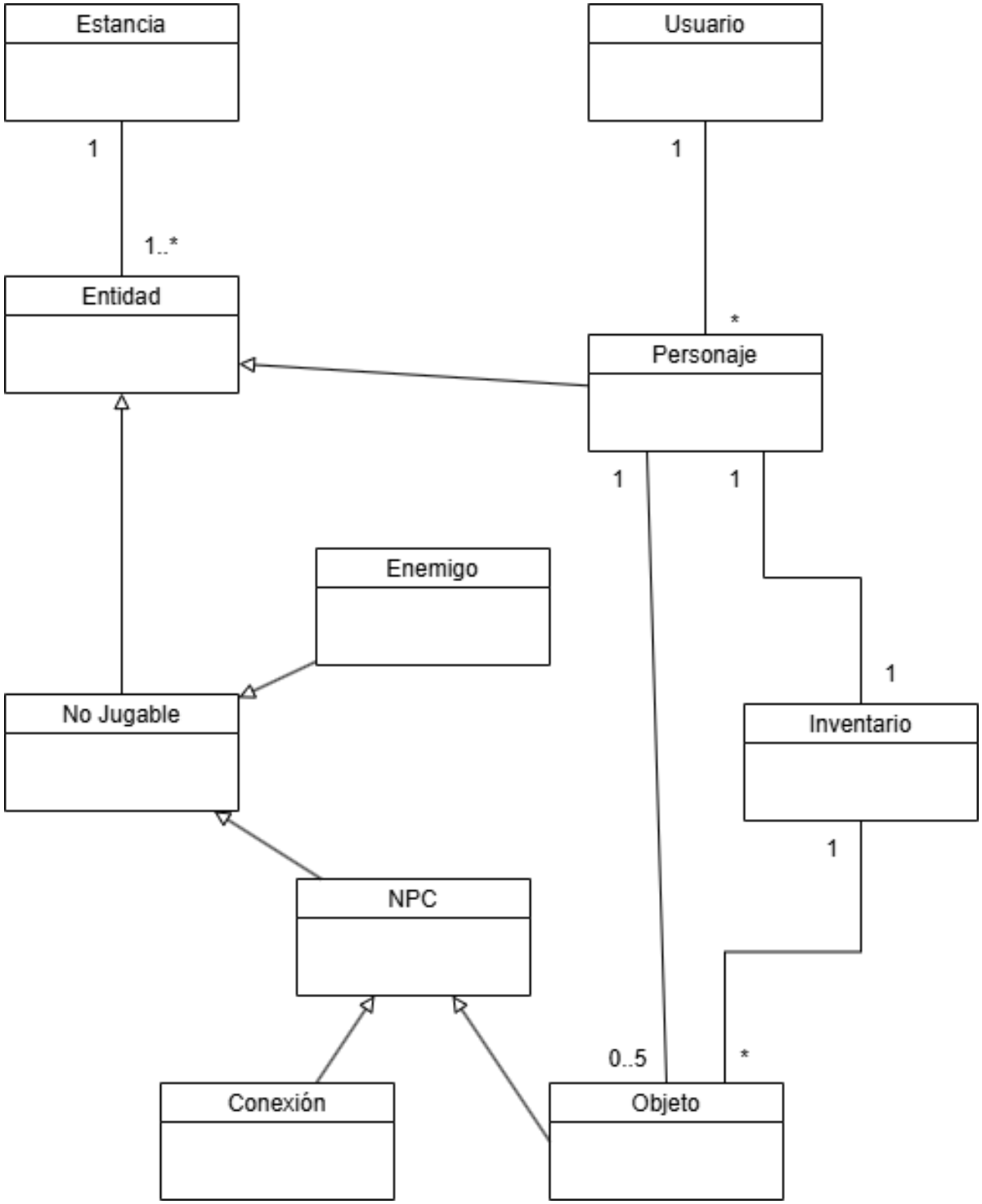

Figura 4.2: Diagrama de análisis Hephaestus

4.2. Juego didáctico

Al tratarse de un foco menor respecto a Hephaestus, el análisis del juego didáctico no es tan extenso y profundo como lo es el del motor. Pese a ello, el análisis es suficiente para que el diseño posterior cumpla con las condiciones suficientes para el correcto funcionamiento de las prácticas.

4.2.1. Requisitos

El juego didáctico se caracteriza por actuar como servidor para clientes creados por los alumnos, mediante los cuales se puede resolver el juego. Es fundamental que se adapte a las necesidades del curso en el que se emplea.

Por lo tanto las siguientes listas muestran los requisitos funcionales y no funcionales del juego didáctico.

Requisito
Tiene que implementar un servidor de tipo TCP que admita comunicaciones concurrentes.
El cliente tiene que poder conectarse empleando el puerto 3000 TCP.
Tiene que admitir, únicamente, comunicaciones que empleen el protocolo JSON-RPC.
Tiene que controlar todos los actores de tipo personaje y enemigo de forma remota.
Tiene que definir las rutas de las máquinas en la que los actores de tipo personaje y enemigo se van a ejecutar.
Tiene que contar con los suficientes procedimientos remotos para ser completado.
Tiene que contar con un procedimiento remoto que devuelva la información correcta para la ejecución del resto de procedimientos.
Toda gestión de la base de datos se tiene que delegar al motor Hephaestus.
Los procedimientos tienen que delegar la gestión de estados de los actores al motor Hephaestus.

Tabla 4.3: Juego didáctico. Requisitos funcionales

Requisito
Tiene que estar escrito en el lenguaje de programación Elixir.
Tiene que hacer uso del motor Hephaestus.

La definición del juego tiene que estar escrita en JSON.
Su longitud tiene que estar adaptada para el contexto de la asignatura en el que es utilizado.
Tiene que desarrollarse sobre un mundo compartido. Esto es, en caso de que se conecten dos personajes, las acciones de uno de los personajes sobre el mundo afecta al mundo que ve el otro personaje, es decir, es el mismo mundo.
Tiene que implementar un procedimiento que ejecute un guardado del estado del juego de manera permanente.
Tiene que implementar un procedimiento para la autenticación de usuarios.

Tabla 4.4: Juego didáctico. Requisitos no funcionales

4.2.2. Modelos

Teniendo en cuenta las características y los requisitos anteriores. El análisis completo del juego didáctico se muestra en el siguiente diagrama.

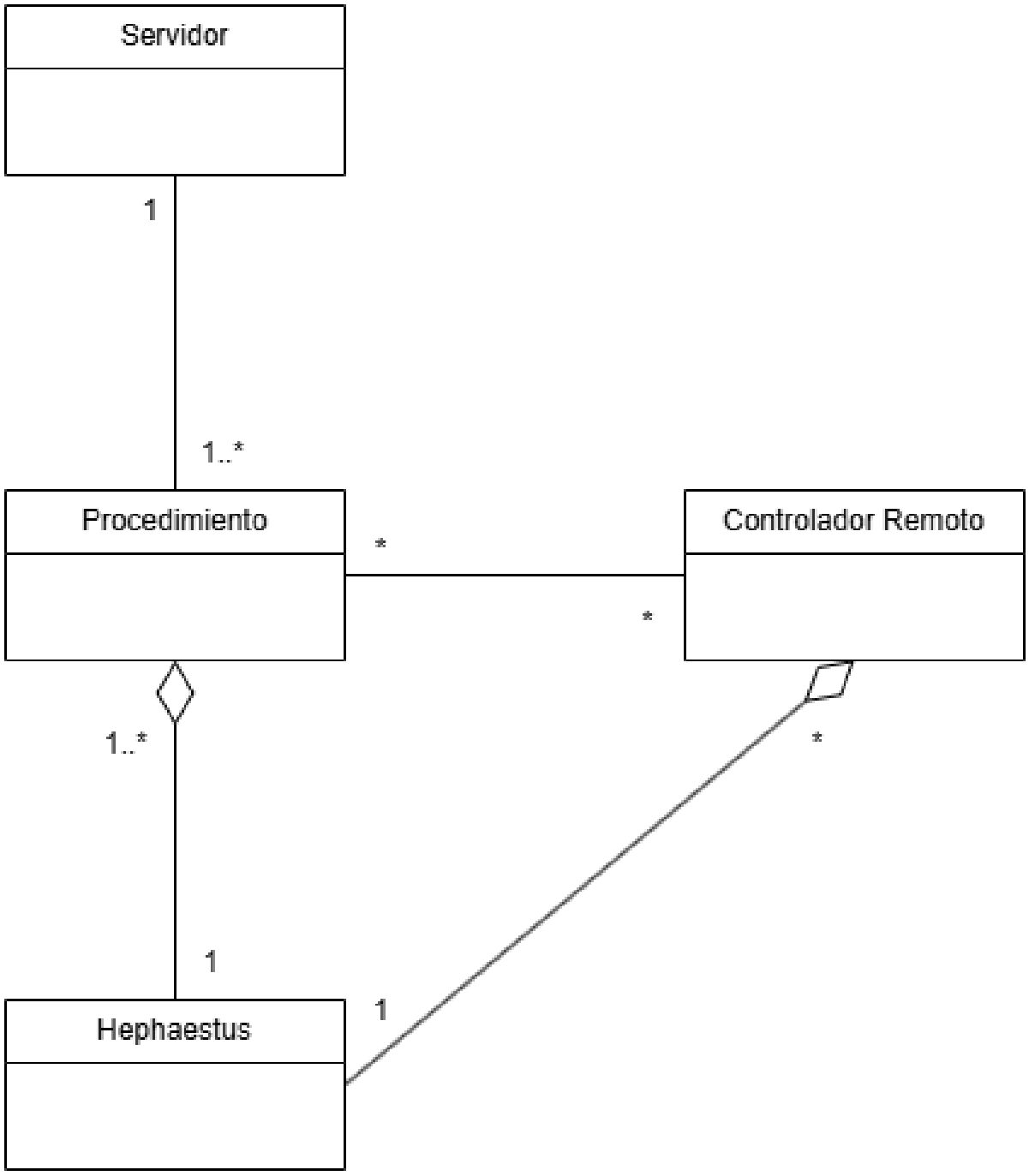

Figura 4.3: Diagrama de análisis juego didáctico

Capítulo 5

Diseño

5.1. Hephaestus

En el capítulo 4 se hizo el análisis del motor de videojuegos Hephaestus, en este se realiza el diseño conceptual del mismo. Para ello se empieza por el diagrama de clases del motor.

5.2. Juego didáctico

5.3. Prácticas

Al contrario que las secciones anteriores, la prácticas para los alumnos no cuentan con un análisis debido a la naturaleza de estas. El objetivo de las prácticas es crear documentos que ayuden al alumno a asimilar el conocimiento adquirido en las clases de teoría. Por lo tanto, el entregable principal de las prácticas para los alumnos son los documentos descriptivos con las instrucciones que los alumnos tienen que seguir para completar las prácticas, incluyendo un reto a completar por el alumno con todo lo aprendido durante la práctica.

Se decide hacer tres prácticas las cuales están relacionadas entre sí, continúan a raíz de la práctica anterior, e incrementan su dificultad siendo la primera la más fácil y la última la más difícil. A estas tres prácticas se añade un cuarto documento que actúa como introducción a las prácticas.

A estas tres prácticas se les añade sus soluciones y otros programas escritos en Java que complementan las prácticas escritas. Todo este código Java está pensado para su uso por el profesor de la asignatura ya sea para tener una solución ya construida para entregar a los alumnos una vez terminado el periodo de la práctica, o para mostrar el funcionamiento de alguna de las partes del enunciado mediante el resto de

programas que no son soluciones.

Por lo tanto, el diseño de las prácticas se trata de una labor de diseño didáctico sin código lo suficiente complejo para justificar un análisis y diseño exhaustivo. A continuación se describe el diseño didáctico de cada una de las prácticas y como se relacionan entre sí.

5.3.1. Introducción a las prácticas

Como ya se ha mencionado se crea un documento diseñado para introducir las tres prácticas al alumno. Para ello el documento cuenta con varias secciones. En estas secciones el alumno cuenta con las instrucciones para instalar todas las tecnologías necesarias para desarrollar las prácticas, las instrucciones necesarias para poner en funcionamiento el juego y un resumen de los objetivos de las prácticas a las que el alumno va a enfrentarse.

Con este documento el alumno es capaz de encontrar soluciones para los problemas más comunes que se pueda encontrar durante el transcurso de las prácticas, como, por ejemplo, levantar los contenedores del proyecto, modificar la base de datos o reiniciar el sistema si surgen algún problema.

5.3.2. Primera práctica

Durante la primera práctica se presenta la base del funcionamiento del sistema. Debido al nivel de los alumnos para los que están pensadas las prácticas, la primera sección de la práctica introduce la arquitectura para sistemas distribuidos cliente-servidor, de esta manera el alumno se introduce en la estructura que van a seguir los clientes que va a programar para resolver las prácticas.

Tras la introducción a la arquitectura distribuida cliente-servidor, la otra pieza fundamental para que el alumno pueda resolver las prácticas es el protocolo JSON-RPC. Por este motivo la siguiente sección de la práctica consiste en una introducción a este protocolo. En esta introducción se añaden ejemplos del formato de las peticiones que admite el protocolo al igual que el formato que sigue el protocolo para las respuesta que se dan en el mismo.

Una vez se hacen las introducciones a los dos conceptos anteriores, la práctica finaliza con el ejercicio a realizar por el alumno. Pero, antes de presentar el enunciado, es necesario realizar un ejemplo de un cliente Java para que el alumno tenga un apoyo visual de lo que se le pide, ilustrado en 5.1. A partir de este cliente, la premisa de esta primera práctica es emular el cliente puesto como ejemplo y, realizar una serie de acciones dentro del juego mediante un solo cliente de Java o varios de ellos.

Con el fin de facilitar al alumno la resolución de la práctica, tanto en esta como en las siguientes prácticas, se proporciona un listado con todas las primitivas necesarias para la resolución de la misma. A su vez, como ya se vio en el capitulo de análisis (4) y en la sección anterior sobre el diseño del juego didáctico,

este cuenta con una primitiva la cual devuelve todas las primitivas disponibles en el juego y como llamarlas.

```
public class JavaClient {
  public static void main(String[] args) throws IOException {
    Gson jsonParser = new Gson();
    Socket socket = new Socket("localhost", 3000);

    // Abrir el canal de comunicacion con el servidor
    PrintWriter out = new PrintWriter(socket.getOutputStream(), true)
       ;
    BufferedReader in = new BufferedReader(new InputStreamReader(
       socket.getInputStream()));

    // Preparar los datos
    HashMap<String, Object> mapa = new HashMap<String, Object>();
    mapa.put("jsonrpc", "2.0");
    mapa.put("method", "info");
    mapa.put("id", "1");

    // Enviar datos
    out.println(jsonParser.toJson(mapa));

    // Obtener respuesta
    HashMap<String, Object> respuesta = jsonParser.fromJson(in.
       readLine(), new TypeToken<HashMap<String, Object>>(){}.getType
       ());

    printServerResult(respuesta); // Imprimir respuesta
    socket.close(); // Cerrar el canal
  }

  private static void printServerResult(HashMap<String, Object> mapa)
     {
    Object resultado = mapa.get("result");
    Object error = mapa.get("error");
    Object id = mapa.get("id");

    if (resultado == null) {
      if (error == null) {
        System.out.println("Algo salio mal en el servidor \n");
```

```
        } else {
          System.out.println("Error al realizar la accion del mensaje
             con id " + id + ": " + error + "\n");
        }
      } else {
        System.out.println("Resultado del mensaje con id " + id + ": "
           + resultado + "\n");
      }
    }
}
```

Código 5.1: Snippet cliente Java entregado a los alumnos

5.3.3. Segunda práctica

En esta segunda práctica se profundiza en uno de los conceptos vistos en teoría, los actores. Además se profundiza en una de las partes de los actores como son las colas de mensajes. Con este objetivo, se comienza la práctica recordando lo visto en teoría sobre los actores, esto ubica a los alumnos en el nuevo contexto y les ayuda a encontrar la información necesaria si lo requieren en la teoría.

Tras la introducción a los actores, la práctica explora en profundidad el concepto de las colas de mensaje, un concepto informático empleado en muchos tipos de sistemas y que, en este caso, forman parte de los actores. Tras estas introducciones se le muestra al alumno cómo se emplean estos conceptos en el desarrollo y contexto del sistema empleado para las prácticas, el motor Hephaestus y el juego didáctico.

Con el contexto de estos conceptos, se prepara la práctica para los alumnos. Al ser la segunda práctica la dificultad de la misma tiene que aumentar y, como ya se ha mencionado, continúa con el hilo de la práctica anterior, jugar al juego didáctico. Para ello se diseña la práctica empleando programación asíncrona. El concepto de la programación asíncrona es un concepto nuevo para el nivel de los alumnos a los que va dirigido el proyecto, por lo tanto, y teniendo en cuenta que ya se ha mostrado en la práctica anterior como se construye un cliente para jugar al juego, se le da al alumno el esqueleto base para poder realizar la práctica mediante programación asíncrona, tal y como se muestra en el código 5.2.

Con todo lo entregado en esta práctica y lo mostrado en la práctica anterior el alumno debe ser capaz de resolver la premisa de la práctica. Los conocimientos que el alumno adquiere de esta práctica son el funcionamiento y uso de los actores y las colas de mensajes, dentro de estos, en un sistema distribuido y la programación asíncrona.

```
private static void interactWithNpc(Gson jsonParser) throws
    IOException, InterruptedException, ExecutionException {
```

```
  // Parametros comunes a todas las llamadas
  HashMap<String, String> params = new HashMap<String, String>();
  params.put("npc", "sabio");
  params.put("character_name", "tim");

  // Ejecutor de tareas asincronas
  ExecutorService executor = Executors.newFixedThreadPool(10);
  // Lista de tareas asincronas
  List<Future<HashMap<String, Object>>> futures = new ArrayList<>();

  // Bucle de creacion y lanzamiento de tareas asincronas
  for (int i = 1; i < 7; i++) {
        HashMap<String, Object> mapa = new HashMap<String, Object>();
        mapa.put("jsonrpc", "2.0");
        mapa.put("method", "interact");
        mapa.put("params", params);
        mapa.put("id", String.valueOf(i));

        Callable<HashMap<String, Object>> task = () -> {
          // Implementacion de un cliente
          return respuesta;
        };

        futures.add(executor.submit(task));
  }

  // Muestra de resultados por pantalla
  for (Future<HashMap<String, Object>> future : futures) {
    printServerResult(future.get());
  }

  executor.shutdown();
}
```

Código 5.2: Snippet codigo asíncrono Java entregado a los alumnos

5.3.4. Tercera práctica

En la tercera y última práctica se diseña una práctica basada no tanto en conceptos vistos en teoría, sino en el desarrollo de sistemas distribuidos y cómo funciona, en concreto, el sistema empleado en las

prácticas.

Con este objetivo la práctica comienza con una introducción a la programación en Elixir, debido al cambio que supone este lenguaje frente a Java. Se explican los fundamentos necesarios para realizar la práctica, esto incluye el funcionamiento del "pattern matching.º las construcciones condicionales, entre otros conceptos.

Una vez se muestra el funcionamiento de Elixir, la práctica continúa con la premisa a resolver por el alumno, en esta práctica dicha premisa consiste en emplear ambos lenguajes para resolver lo planteado. En primer lugar se emplea Elixir para modificar el motor del juego Hephaestus y, en segundo lugar, se emplea Java para completar el juego, dando por finalizada la práctica.

Mediante este diseño de la práctica se incrementa la dificultad de la misma, mientras que se muestra el funcionamiento de un sistema distribuido y se continúa con el hilo de todas estas prácticas que es la finalización del juego didáctico.

Capítulo 6

Pruebas

La gestión de la calidad del software se asegura que el desarrollo de los sistemas software desarrollados se encuentran en un estado correcto para el propósito para el que fueron diseñados. Esto implica que el sistema debe asegurar las necesidades de los clientes que tiene y ser eficiente y fiable durante su periodo de funcionamiento [6].

Existen diferentes tipos de pruebas para determinar la calidad del software, las cuales se enfocan en diferentes propiedades de un sistema. De esta manera se puede asegurar que el sistema no contenga errores que impidan su utilización. Además, existen técnicas que permiten asegurar que cada parte del sistema funcione correctamente sin producir resultados inesperados. La división de estas partes puede ser a nivel más general o más concreta, siendo por ejemplo una parte general un componente del sistema y una parte más concreta una función dentro de un componente del sistema.

Durante el proyecto se emplean tres tipos diferentes de pruebas para los distintos entregables. Debido al tiempo disponible, el rango de actuación y la cantidad de tipos de pruebas realizadas a cada una de las partes se ve disminuido. Por ello se prioriza el correcto funcionamiento del sistema y una experiencia como mínimo adecuada para el alumno, se deja de lado pruebas más profundas y se centra el mayor esfuerzo en el motor, como pieza central del proyecto.

La cantidad de pruebas y profundidad de las mismas se ve afectado por el tiempo disponible. Por ello y debido a que la posibilidad de recuperar el sistema y dejarlo en un estado anterior es muy sencillo, gracias al empleo de Docker, se hicieron recortes en este punto del proyecto, siendo las pruebas realizadas suficientes para asegurar el correcto funcionamiento del sistema, pero dejando de lado la integridad del mismo.

6.1. Pruebas unitarias

Las pruebas unitarias en el desarrollo de software sirven para verificar el funcionamiento de las partes más pequeñas del sistema que está siendo probado. Se aseguran de que las partes verificadas se comporten como se espera y no surjan ningún error.

Por lo general las pruebas unitarias requieren de un tiempo mayor a otro tipo de pruebas para su implementación ya que, al centrarse en las partes más pequeñas del sistema evaluado, requieren de múltiples pruebas por unidad del sistema de manera que se abarquen todos los escenarios posibles.

Debido a su naturaleza, las pruebas unitarias son muy útiles para probar sistemas o aplicaciones software que normalmente no cuentan con interacción directa del usuario, sino que son empleados por otros sistemas, por ejemplo las bibliotecas de algún lenguaje. Por ello, y debido al recorte por tiempo ya mencionado, estas pruebas solo se emplean para asegurar el correcto funcionamiento del motor Hephaestus.

Al tratarse de Hephaestus el lenguaje para las pruebas del motor es Elixir, por lo tanto se hace uso de las herramientas proporcionadas por el propio lenguaje para hacer las pruebas, tal y como se muestra en el ejemplo 6.1. Mediante estas pruebas se asegura el correcto funcionamiento de las piezas del motor necesarias para crear un videojuego, incluyendo todo lo implementado mediante *GenServer*, para lo cual las pruebas se centran en el código que produce algún cambio en el estado, esto es se excluye de las pruebas todo código que no modifique el comportamiento de la biblioteca *GenServer* como puede ser un *getter* de un campo del estado.

```
defmodule ConnectionTest do
  use ExUnit.Case, async: true

  setup do
    connection =
    start_supervised!(
      {Engine.Connection,
        %Engine.GameEntity{
              name: "test_connection",
              state: %Engine.Connection{
                    level: 1,
                    location_1: "location_test",
                    location_2: "second_location_test",
                    object: nil
              }
      }}
    )
```

```
    %{test_connection: connection}
  end

  test "get location from location_1",
       %{test_connection: test_connection} do
    assert Engine.Connection.get_next_location(test_connection,
           "location_test") === "second_location_test"
  end
end
```

Código 6.1: Snippet test unitario Elixir

6.2. Pruebas end-to-end

Las pruebas end-to-end simulan el flujo completo del sistema desde la perspectiva del usuario final. De esta manera se prueba la experiencia de usuario obtenida al interactuar con el sistema. Por lo tanto estas pruebas son las más adecuadas para probar el juego didáctico, puesto que el alumno, el cual actúa como usuario final en este proyecto, interactúa con dicha parte del proyecto.

Para la realización de las pruebas end-to-end se crean una serie de escenarios los cuales el juego debe cumplir para poder ser considerado exitoso. Dichos escenarios derivan, más adelante en el desarrollo del proyecto, en las soluciones a las prácticas presentadas a los alumnos, puesto que el juego está diseñado específicamente para cumplir con ese propósito.

El juego didáctico actúa como hilo conductor de las prácticas y, por lo tanto, es natural que se enlace las pruebas end-to-end del mismo con las prácticas proporcionadas a los alumnos, en este caso como solución de las mismas.

Capítulo 7

Conclusión

Una vez finalizado el proyecto, se analizan los problemas encontrados durante el desarrollo del mismo y el desarrollo futuro del mismo tras el estado en el que termina con este documento.

7.1. Problemas en la implementación del código

Una vez se concluye el análisis y el diseño del proyecto se procede a la implementación del código de acuerdo a lo diseñado. Durante este proceso, como en todo proceso de implementación, se encuentran varios problemas los cuales se tienen que solucionar.

El proyecto cuenta con varias partes de las cuales dos de ellas están escritas en Elixir mientras que la última está escrita en Java. Ya se ha mencionado en capítulos anteriores que el código escrito en Java forma parte de las soluciones para las prácticas, por lo tanto es código sencillo que no requiere de análisis y diseños profundos y que, además, no supone un problema su implementación.

Por lo tanto, los focos de problemas durante la implementación del código se concentran en Elixir y Docker. Elixir supone un reto debido a la novedad del lenguaje para el alumno, mientras que el reto que supone Docker deriva de Elixir y la construcción de las imágenes para la ejecución de programas en este lenguaje.

7.1.1. Elixir

Al ser Elixir un lenguaje de programación que sigue el paradigma de programación funcional y no cuenta con herramientas para la programación orientada a objetos. Estas dos características del diseño del lenguaje desarrollan los dos primeros problemas a la hora de implementar el código. El cambio de paradigma del imperativo al funcional supone el mayor coste temporal al proyecto, debido a que las

diferencias son demasiado pronunciadas. Respecto a la programación orientada a objetos, se adapta el código empleando los elementos que ofrece el lenguaje, esto no supone tanto costo temporal al proyecto.

El siguiente elemento del lenguaje que se presenta problemático son los *GenServer*. Estos elementos presentan unas características que los hacen buenos candidatos a representar actores distribuidos dentro del sistema. Tras algunas vueltas se acaba por decidir por ellos para hacer esta labor, además de ser la solución a la adaptación del problema de la programación orientada a objetos. Debido a las características de los *GenServer* pueden actuar como una especie de objetos tradicionales, de esta manera se consigue una simulación parcial, no pudiendo conseguir la composición y la herencia características de este paradigma. Por lo tanto no se puede evitar cierta repetición de código.

Una vez se superan estos problemas ya descritos, el siguiente problema que surge es la transformación de estructuras de Elixir a JSON. Debido a que el estado del juego se almacena como JSON en la base de datos, es fundamental poder transformar estructuras de Elixir a JSON. Elixir cuenta con mapas como estructura para el almacenamiento de datos en tiempo de ejecución, estos mapas tienen transformación directa a JSON al emplear la biblioteca con la que se hace uso de la base de datos desde el motor. Por lo tanto es un problema con fácil solución y que no supone ningún retraso.

Finalmente, el último problema que se presenta por usar Elixir son los monitores. Los monitores son un elemento de Elixir que sirven para vigilar la ejecución de procesos hijos, en el caso de Hephaestus se emplean para vigilar la ejecución de los *GenServer*. Los monitores no solo vigilan los procesos sino que actúan sobre ellos en caso de que se produzca un error, pudiendo reiniciarlos o eliminarlos dependiendo de las instrucciones que tengan. El problema surge de los diferentes tipos de monitores que existen y de como se relacionan entre sí. Se evalúan todas las opciones y se opta por emplear monitores dinámicos por la sencillez que supone su uso. Este tipo de monitores es adecuado para juegos pequeños como lo es el juego didáctico, pero para su uso en juegos más grandes el desarrollador debe evaluar si merece la pena emplear monitores estáticos, los cuales permiten más opciones.

7.1.2. Docker

Los problemas surgidos por el empleo de Docker son derivados de Elixir. La distribución de Elixir difiere en gran medida del proceso de desarrollo, por lo tanto la construcción de las imágenes es más complicado de lo que sería en caso de ser un entorno de desarrollo. Otro factor determinante es la falta de documentación, Elixir es un lenguaje relativamente pequeño en lo que respecta a comunidad y, por lo tanto, hay ciertas tareas que no son tan populares y que no cuentan con la suficiente documentación para su ejecución.

En este caso la solución pasa por adaptar los pocos ejemplos existentes, sin relación con la tarea específica, y emplear herramientas de inteligencia artificial para generar el código restante. Finalmente, se hace uso de estos métodos para solucionar el problema, costando bastante tiempo de proyecto.

7.2. Desarrollo a futuro

Una vez se finaliza el desarrollo del proyecto descrito en este documento dicho proyecto puede seguir siendo desarrollado. Para ello se presentan tareas o partes enteras del proyecto que no se han podido llegar a realizar. Mediante este listado, cualquier futuro desarrollador cuenta con una imagen global de lo que falta, así como de ideas lanzadas que puedan ser interesantes.

7.2.1. Mejora en el proceso de pruebas

Como ya se ha mencionado en el capítulo 6, debido a la falta de tiempo, las pruebas realizadas sobre el motor Hephaestus y el juego didáctico solamente aseguran el correcto funcionamiento de estos.

Con el fin de mejorar la calidad del código se plantea la introducción de un mayor número de pruebas unitarias que mejoren la integridad de Hephaestus, así como la integración de este tipo de pruebas en el juego didáctico.

7.2.2. Ampliación del motor

Desde el principio se ha declarado que el motor debe ser ampliable y tener la capacidad de añadir más módulos para crear juegos más complejos o incluir una mayor cantidad de utilidades para los desarrolladores.

En el capítulo 4 se muestra una serie de elementos que fueron descartados por la falta de tiempo. Estos elementos se pueden ver en la figura 4.1 y son un buen punto de inicio para una futura expansión del motor Hephaestus.

7.2.3. Separación en motores más pequeños

Aunque el motor se ha diseñado para ser distribuido y actuar como base de ejecución para diferentes piezas en diferentes máquinas, existe la posibilidad de especializar aún más esta labor.

Se propone la división del propio motor en motores más pequeños con los módulos del motor original necesarios para ejecutar una pieza o piezas específicas en cada máquina.

Este aumento de la complejidad de la arquitectura y la mantenibilidad del código solo es justificable si el tamaño del motor y los recursos necesarios para hacerlo funcionar aumentan, de tal manera que su ejecución en cada máquina suponga un coste operacional tan alto que desincentive su uso en sistemas distribuidos.

7.2.4. Generalizar y aumentar la especialidad del motor

Al final del proyecto, el motor Hephaestus es un motor diseñado para crear videojuegos distribuidos de tipo RPG basados en Dungeons and Dragons. Se trata de una especialización muy concreta que reduce el mercado en el que se puede emplear dicho motor.

Una futura modificación del motor puede generalizar tanto el género de juegos RPG al que afecta, no solo a los que están basados en Dungeons and Dragons, cómo el tipo de juego que se puede construir con dicho motor. De esta manera se avanza hacia un motor más generalista como pueden ser Godot, Unity o Unreal Engine.

Bibliografía

[1] Firmantes del manifiesto. Manifesto for agile software development. https://agilemanifesto.org/, 2001. [Online; último acceso 27-Diciembre-2025].

[2] Bob Hughes y Mike Cotterell. *Software Project Management*. McGraw Hill, 2009.

[3] El equipo de Elixir. Elixir ejemplo servidor tcp. https://hexdocs.pm/elixir/task-and-gen-tcp.html#echo-server, 2025. [Online; último acceso 28-Diciembre-2025].

[4] mtrudel. Biblioteca thousand island. https://hexdocs.pm/thousand_island/ThousandIsland.html, 2025. [Online; último acceso 28-Diciembre-2025].

[5] docsifyjs. Biblioteca docsify. https://docsify.js.org/#/, 2025. [Online; último acceso 28-Diciembre-2025].

[6] Ian Sommerville. *Software engineering 10th Edition*. Pearson, 2016.

[7] Ministerio de Trabajo y Economía Socila, Gobierno de España. El boe publica el smi para 2025 que se establece en 1.184 euros. https://www.sepe.es/HomeSepe/es/que-es-el-sepe/comunicacion-institucional/noticias/detalle-noticia.html?folder=/SEPE/2025/Febrero/&detail=boe-publica-smi-2025-se-establece-1184-euros, 2025. [Online; último acceso 31-Diciembre-2025].

[8] Taxfix. Salario mínimo interprofesional (smi). https://taxfix.com/es-es/diccionario/salario-minimo-interprofesional-smi/#:~:text=Gracias%20al%20acuerdo%20en%202025%20del%20Gobierno,trabajadores%20a%20jornada%20completa%20(40%20horas%20semanales)., 2025. [Online; último acceso 31-Diciembre-2025].

[9] Wikipedia. Second life. https://en.wikipedia.org/wiki/Second_Life, 2025. [Online; último acceso 31-Diciembre-2025].

[10] David Alexander Bond. Design and implementation of a massively multi-player online historical role-playing game. *MSc Advanced Internet Applications. Heriot-Watt School of Mathematical and Computer Sciences*, 2015.

[11] El equipo de Elixir. Elixir. `https://elixir-lang.org/`, 2025. [Online; último acceso 21-Junio-2025].

[12] El equipo de Phoenix. Phoenix framework. `https://www.phoenixframework.org/`, 2025. [Online; último acceso 01-Julio-2025].

[13] Oracle. Java. `https://www.java.com/es/`, 2025. [Online; último acceso 01-Julio-2025].

[14] Wikipedia. Sql. `https://en.wikipedia.org/wiki/SQL`, 2025. [Online; último acceso 01-Julio-2025].

[15] Douglas Crockford. Json. `https://www.json.org/json-en.html`, 2001. [Online; último acceso 01-Julio-2025].

[16] Mozilla foundation. Javascript. `https://developer.mozilla.org/es/docs/Web/JavaScript`, 2025. [Online; último acceso 01-Julio-2025].

[17] El equipo de Postgresql. Postgresql. `https://www.postgresql.org/`, 2025. [Online; último acceso 01-Julio-2025].

[18] El equipo de Docker. Docker. `https://www.docker.com/`, 2025. [Online; último acceso 01-Julio-2025].

[19] The linux foundation. Kubernetes. `https://kubernetes.io/es/`, 2025. [Online; último acceso 01-Julio-2025].

[20] Matt Morley. Json-rpc. `https://www.jsonrpc.org/`, 2005. [Online; último acceso 03-Enero-2026].