

UNIVERSIDAD DE SANTIAGO DE COMPOSTELA
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA



Inteligencia Artificial aplicada a Videojuegos Top-Down en tiempo real

TRABAJO DE FIN DE GRADO

Autor:

Rubén Osorio López

Tutores:

Manuel Mucientes Molina

Pablo Rodríguez Mier



D. Manuel Mucientes Molina, Profesor del Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela, y **D. Pablo Rodríguez Mier**, Profesor del Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela,

INFORMAN:

Que la presente memoria, titulada *Inteligencia Artificial aplicada a Videojuegos Top-Down en tiempo real*, presentada por **D. Rubén Osorio López** para superar los créditos correspondientes al Trabajo de Fin de Grado de la titulación de Grado en Ingeniería Informática, se ha realizado bajo nuestra tutoría en el Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela.

Y para que así conste a los efectos oportunos, expiden el presente informe en Santiago de Compostela a (Data):

El tutor,

El cotutor,

El alumno,

Manuel Mucientes Molina Pablo Rodríguez Mier Rubén Osorio López

Agradecimientos

A mi tutor Manuel y a mi cotutor Pablo por su inestimable e imprescindible ayuda durante el periodo de desarrollo de este trabajo.

A mis compañeros de clase que me han sabido animar cuando más lo necesitaba.

A mi familia por apoyarme durante todo este tiempo.

Índice general

1. Prefacio	1
1.1. Objetivos	2
1.2. Organización del documento	3
2. Introducción	5
3. Análisis de requisitos	7
3.1. Identificación de los requisitos del sistema	7
3.1.1. Actores	8
3.1.2. Casos de uso	8
3.1.3. Especificación de requisitos	13
4. Gestión del proyecto	37
4.1. Gestión de riesgos	37
4.1.1. Especificación de riesgos	39
4.1.2. Control de riesgos	47
4.2. Gestión de la configuración	51
4.2.1. Herramientas	52
4.2.2. Descripción del soporte de las herramientas al proceso . .	52
4.2.3. Nomenclatura	53
4.3. Metodología de desarrollo	54
4.4. Planificación temporal	56
4.4.1. Planificación del videojuego	58
4.4.2. Planificación del agente	60
4.4.3. Planificación de la obtención de datos	61
4.4.4. Planificación de la documentación, reuniones y finalización	61
4.5. Análisis de costes	63
4.6. Plan de Gestión de las Comunicaciones	66
4.6.1. Gestión de interesados	66
4.6.2. Planificación de la información y comunicaciones	66
4.6.3. Reuniones	67

5. Arquitectura y herramientas	71
5.1. Arquitectura del sistema	71
5.1.1. Subsistemas propios del motor	75
5.2. Herramientas de diseño	76
5.2.1. Herramientas software	76
5.2.2. Patrones de diseño	76
5.3. Herramientas de desarrollo	81
5.3.1. Realización de la documentación	82
5.3.2. Desarrollo	82
5.3.3. Tecnologías	83
5.3.4. Librerías	84
6. Diseño e implementación	85
6.1. Diagramas de clases	86
6.1.1. Bus de mensajes	86
6.1.2. Aplicación general	87
6.1.3. Subsistemas	87
6.1.4. Otros útiles	94
6.1.5. Escenas	97
6.2. Diagramas de secuencia	103
6.2.1. Aplicación general	103
6.2.2. Escenas y agente	108
6.3. Diseño de la interfaz gráfica	113
6.3.1. Interfaz del menú	113
6.3.2. Interfaz del combate	114
6.3.3. Interfaz de la consola	115
7. Validación y pruebas	117
8. Valoraciones finales	119
9. Ejemplos	123
9.1. Un ejemplo de sección	123
9.1.1. Un ejemplo de subsección	123
9.1.2. Otro ejemplo de subsección	123
9.2. Ejemplos de figuras e cuadros	124
9.3. Ejemplos de referencias á bibliografía	124
9.4. Ejemplos de enumeracións	125
10. Conclusiones y posibles ampliaciones	127
A. Manuais técnicos	129

B. Manuais de usuario	131
C. Licencia	133
Bibliografía	135

Índice de figuras

3.1. Diagrama de casos de uso	10
4.1. Estructura de Descomposición del Trabajo o EDT	57
4.2. Cronograma general del proyecto	58
4.3. Cronograma de la parte del videojuego	59
4.4. Cronograma de la parte del agente	60
4.5. Cronograma de la parte de obtención de datos	61
4.6. Cronograma de la parte de documentación, reuniones y finalización	62
5.1. Arquitectura con subsistemas interconectados directamente . . .	72
5.2. Arquitectura con subsistemas interconectados mediante bus de mensajes	73
5.3. Arquitectura con Bus Node	74
5.4. Arquitectura final	74
5.5. Patrón <i>Strategy</i>	78
5.6. Patrón <i>Singleton</i>	79
5.7. Patrón <i>Decorator</i>	80
5.8. Patrón Publica-Subscribe o <i>PubSub</i>	80
6.1. Diagrama de clases del bus de mensajes	87
6.2. Diagrama de clases de la lógica del juego	88
6.3. Diagrama de clases del subsistema de consola	89
6.4. Diagrama de clases del subsistema de entrada	90
6.5. Diagrama de clases del subsistema de físicas	91
6.6. Diagrama de clases del subsistema de renderizado	93
6.7. Diagrama de clases del subsistema de sonido	95
6.8. Diagrama de clases del paquete de recursos	95
6.9. Diagrama de clases del paquete de útiles	96
6.10. Diagrama de clases de la escena del menú	97
6.11. Diagrama de clases de la escena de juego	98
6.12. Diagrama de clases del comportamiento del agente	102
6.13. Diagrama de secuencia general de la aplicación	104

6.14. Diagrama de secuencia genérico de una escena	106
6.15. Diagrama de secuencia de la consola	107
6.16. Diagrama de secuencia de la escena del menú	109
6.17. Diagrama de secuencia de la escena del <i>gameplay</i>	111
6.18. Diagrama de secuencia del agente	112
6.19. <i>Mockup</i> de la interfaz del menú	114
6.20. <i>Mockup</i> de la interfaz del combate	115
6.21. <i>Mockup</i> de la interfaz de la consola	116
9.1. Esta é a figura de tal e cal.	124

Índice de cuadros

3.1. Matriz de trazabilidad de requisitos contra casos de uso	36
4.1. Valores posibles de probabilidad e impacto	39
4.2. Acción correctiva sobre la imposibilidad de realizar simulaciones aceleradas	48
4.3. Tabla de salario del desarrollador	64
4.4. Tabla de costes de amortización del equipo	64
4.5. Tabla de costes finales	65
4.6. Matriz de interesados del proyecto	68
4.7. Matriz de comunicaciones del proyecto	69
9.1. Esta é a táboa de tal e cal.	124

Capítulo 1

Prefacio

Los juegos de lógica o estrategia han sido utilizados como *benchmark* en el campo de la inteligencia artificial desde sus inicios. Pese a que principalmente se han utilizado juegos de mesa o de cartas dadas las facilidades a la hora de representar su modelo, con el avance de la tecnología y la llegada de los videojuegos en ordenador se consideró la posibilidad de que los mismos podrían ser una nueva herramienta con la que experimentar y explorar diferentes técnicas e implementaciones de IA. En la actualidad prácticamente todos los videojuegos que incorporan elementos que simulan comportamiento humano lo hacen con lógica determinista y patrones definidos, sin incorporar ningún tipo de simulación de pensamiento complejo. Algunas de las razones causantes de este fenómeno son:

- La diferencia en la dificultad de implementación entre patrones simples y un agente complejo.
- El hecho de que un agente puede llegar a ser demasiado efectivo a la hora de competir contra un humano.
- Limitaciones en la capacidad de computación restante cuando el sistema está ejecutando un videojuego ya que suelen ser aplicaciones significativamente pesadas y complejas.

Sin embargo, estos problemas son solventables y debemos tener en cuenta los beneficios que se pueden obtener al trabajar con videojuegos reales. En la actualidad, somos capaces de simular grandes mundos con millones de entidades y jugadores reales. Estos entornos comúnmente utilizan simulaciones físicas realistas y trabajan con datos numéricos continuos en lugar de acciones discretas, es decir, una entidad que simule estar viva en dicho entorno se podría mover cara una posición concreta y realizar una determinada acción durante un periodo de

tiempo sin necesidad de que ninguno de estos conceptos esté atado a una representación discreta. Nos encontramos el caso contrario con los juegos de mesa o cartas tradicionales ya que en estos solo existen acciones discretas y simbólicas, no se trata con ningún tipo de estado continuo.

Por estas razones se podría considerar que un videojuego está mucho mas cerca de ser capaz de modelar el mundo real que un juego de mesa, lo que lo hace un banco de trabajo idóneo para investigar y experimentar con Inteligencia Artificial. Técnicas como redes neuronales, computación evolutiva o lógica difusa pueden funcionar en situaciones en tiempo real, continuas y complejas que presentan tanto nuestra realidad como los videojuegos. Por esto se pretende investigar y experimentar sobre cuales de estas técnicas pueden ser de utilidad en este tipo de situaciones para lo cual se implementará un agente que compita con un jugador real en un pequeño videojuego en el que ambos se moverán libremente en un espacio de posiciones continuo y realizarán una serie de acciones ofensivas y defensivas con el objetivo de derrotar a su oponente.

Con este fin, se pretende realizar una implementación de un juego que permita al agente tener acceso a los estados en los que se encuentran el y su competidor así como las acciones que puede realizar en un determinado momento. Todo esto con la finalidad de averiguar cuales de las posibles técnicas de IA pueden ser usadas para lograr un competidor no solo apto pero también justo. Además, esto nos permitirá realizar un entrenamiento en el que el propio agente podrá competir con él mismo para aprender los fundamentos del juego además de competir contra jugadores humanos. Esto nos brinda la posibilidad de determinar como reacciona ante estos y si es capaz de adaptarse a los posiblemente distintos estilos de juego contra los que se tendrá que enfrentar.

1.1. Objetivos

El objetivo global es implementar un agente que sea capaz de competir contra un jugador simulando un comportamiento humano para lo cual se realizará una pequeña demostración de un videojuego y la implementación del agente con las técnicas que resulten ser más eficaces a la hora de lograr el mencionado objetivo.

Los sub-objetivos que tendremos que abarcar son los siguientes:

1. **Implementar la demostración del videojuego:** Se necesita una plataforma que permita tanto a un jugador humano como al agente interactuar con el entorno del videojuego siguiendo ambos las reglas que este mismo define.

2. **Crear la librería que implemente nuestro agente con las técnicas escogidas:** Se requiere realizar una implementación del agente utilizando las técnicas de inteligencia artificial que hayamos escogido para simular un comportamiento aparentemente humano.
3. **Realizar el entrenamiento del agente:** Una vez realizada la implementación del agente se necesitará ejecutar un proceso de entrenamiento para que este adquiriera la información necesaria para comportarse adecuadamente en el entorno competitivo que el juego presenta.
4. **Obtener datos sobre las capacidades del agente:** Se deberán obtener datos sobre el comportamiento del agente en el entorno del juego al competir con otras posibles implementaciones del mismo que no incluyan el uso de técnicas de inteligencia artificial.
5. **Analizar los resultados obtenidos** Haciendo una recopilación de la información obtenida durante las etapas de diseño, implementación y obtención de datos se realizará un análisis que resuma lo que ha logrado el agente centrándose en aspectos como el desafío que es capaz de presentar, si es capaz de simular a un humano y la posibilidad de usar implementaciones similares en videojuegos en un futuro.

1.2. Organización del documento

La finalidad de este documento es presentar como se han resuelto los objetivos definidos para el proyecto. Para ello se explicarán las diferentes partes que forman el producto final así como las tareas que han sido realizadas a lo largo del proyecto y que han dado lugar al mismo tal y como se presenta.

Completar
la orga-
nización
del do-
cumento

Capítulo 2

Introducción

Capítulo 3

Análisis de requisitos

Referenciando directamente al PMBOK [1] se define la recopilación de requisitos como:

“[...]el proceso que consiste en definir y documentar las necesidades de los interesados a fin de cumplir con los objetivos del proyecto. El éxito del proyecto depende directamente del cuidado que se tenga en obtener y gestionar los requisitos del proyecto y del producto”

Como bien se indica en la cita anterior, el éxito de un proyecto está estrechamente relacionado con la calidad y precisión empleados en el proceso de recogida, análisis y gestión de los requisitos del mismo. Esto hace que imperiosa la necesidad de llevar a cabo los procesos mencionados con extrema minuciosidad y cuidado.

A Continuación se estudiarán las necesidades del proyecto en términos de funcionalidades y objetivos deseables para así obtener una idea sobre el alcance del mismo. Para ello se realizará la identificación y descripción de los múltiples requisitos que se deberán cumplir, separándolos según su índole y relacionándolos con los casos de uso apropiados.

3.1. Identificación de los requisitos del sistema

El primer paso a la hora de realizar correctamente el proceso de análisis de requisitos es dedicar el tiempo necesario para comprender cuales son los objetivos y alcance reales del mismo. Para ello se han llevado a cabo varias reuniones con los tutores previas a la reunión del proyecto en las que se ha discutido cual sería el producto final buscado así como los requisitos que dicho producto tendría que cumplir para estar a la altura de dicho ideal. En sucesivas reuniones iterativas

con ambos tutores se ha logrado tener una idea común del alcance a cumplir mediante el refinamiento de los requisitos, buscando la mínima ambigüedad posible en su definición.

3.1.1. Actores

El primer paso que se debe afrontar dentro del proceso de análisis de requisitos es la identificación de los actores que intervendrán en la ejecución del sistema de una forma u otra. Un actor representa cualquier tipo de rol interpretado por una persona, dispositivo o sistema externo que tendrá capacidad de interactuar con nuestro producto. En este caso particular, la definición de los actores es sencilla y directa: el único rol que se debe definir es el del **usuario que interacciona con el videojuego** descrito en esta memoria. En aras de hacer una especificación formaremos realizaremos la descripción de los actores en la siguiente tabla:

ID	Actor-01
Nombre	Jugador
Descripción	Este actor representa al usuario que interaccionará con el videojuego, realizando partidas contra otro jugador o el agente implementado así como ejecutando el proceso de simulación de partidas que el agente usará como proceso de aprendizaje.

3.1.2. Casos de uso

Una vez especificados los actores que interactúan con el sistema el siguiente paso es identificar y especificar los diferentes casos de uso. Dichos casos de uso nos permitirán centrarnos luego en los requisitos del sistema partiendo de una base sólida.

Se definirá caso de uso como [3] una secuencia de acciones, incluyendo variantes, que ejecuta un sistema para producir un resultado observable de valor para un actor. De esta forma se pueden utilizar los casos de uso para definir el comportamiento deseado durante la etapa de desarrollo realizando una abstracción que permitirá que los miembros involucrados en el proyecto puedan hacer referencia a ellos sin preocuparse de detalles de bajo nivel. De esta forma se facilitarán discusiones sobre el funcionamiento deseado del proyecto, incluido el análisis de requisitos.

De este mismo modo, lo que se hará a continuación es reconocer las diferentes acciones que pueden llevar a cabo los actores, en este caso solamente uno, sobre el sistema. Haciendo una primera aproximación a esta definición de un modo informal podemos considerar las siguientes acciones: Un usuario deberá ser capaz de seleccionar entre las diferentes acciones a realizar dentro del entorno del juego, lo que dará lugar a la ejecución de otros casos de uso diferentes (**seleccionar la opción a ejecutar dentro del videojuego**). Las diferentes opciones darán lugar a casos de uso diferentes ya que el usuario podrá competir con otro usuario con el mismo rol (**competición de jugador humano contra jugador humano**) o competir con el agente entrenado (**competición de jugador humano contra agente**). Finalmente se podrá hacer que el agente compita con otra versión de sí mismo (**competición de agente contra agente**). Además esta última acción se puede llevar a cabo sin que el actor visualice el proceso exponiendo solo los resultados con el fin de llevar a cabo el proceso de aprendizaje del agente.

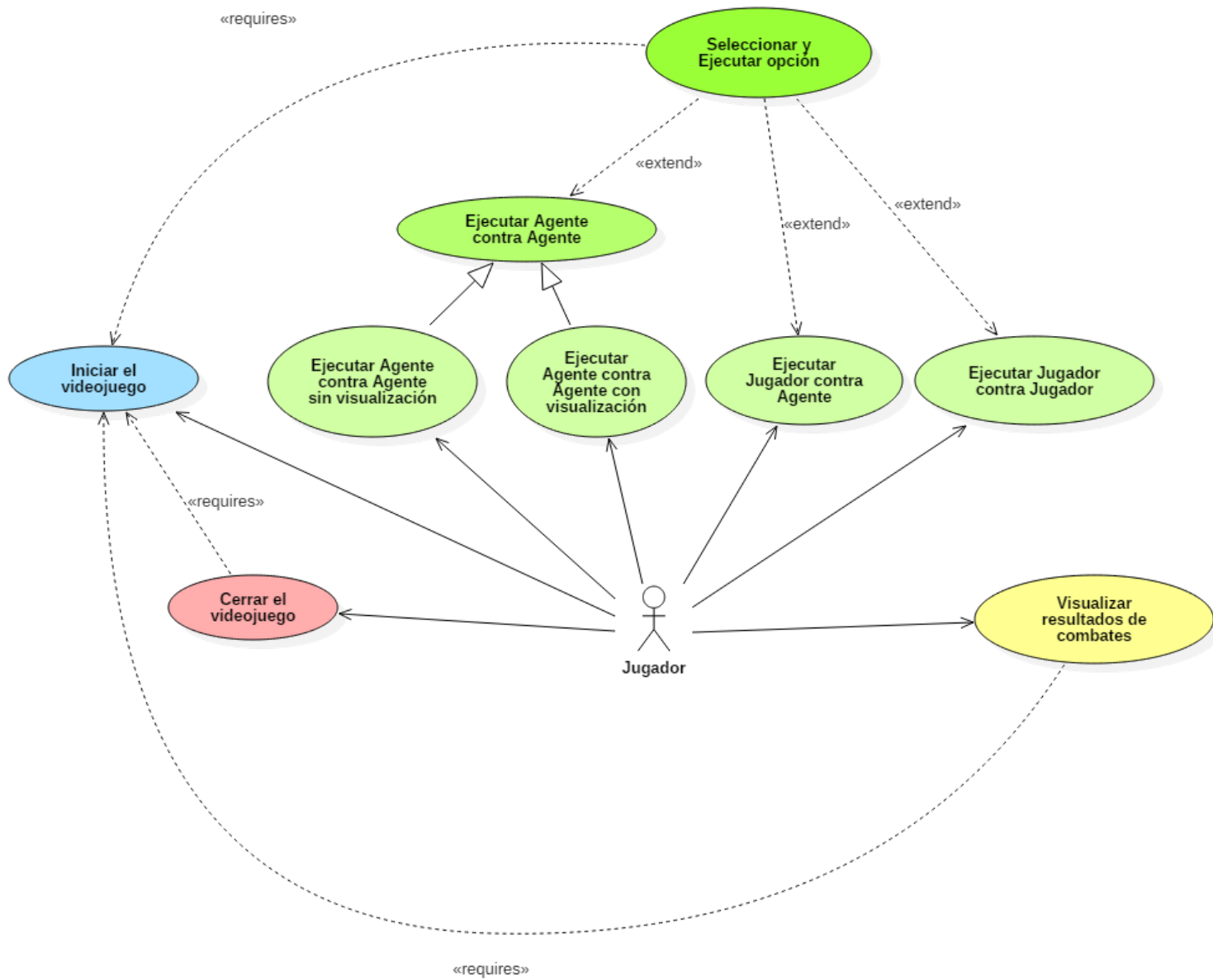


Figura 3.1: Diagrama de casos de uso

En la figura 3.1 se muestra el diagrama de casos de uso a partir del cual podremos realizar una definición formal de los mismos. En dicha figura se han agrupado por colores los casos de uso con características similares para facilitar su comprensión y visualización. En este caso se utilizan el color azul y rojo para abrir y cerrar el programa, colores verdes para los diferentes tipos de ejecuciones posibles del combate y el amarillo para ver los resultados de los combates realizados en una ejecución.

A continuación se muestran los casos de uso especificados de manera formal. La notación utilizada para la identificación de los diferentes casos de uso será **CU¹-N²**. Los campos presentes en cada descripción contarán con un identificador, nombre, descripción y una prioridad. El nivel de prioridad podrá tener uno de los siguientes tres valores con su significado asociado:

- **Vital:** Es fundamental para completar el proyecto.
- **Importante:** Aumenta de forma significativa la calidad y funcionamiento del proyecto pero no tiene la cualidad de indispensable.
- **Deseable:** Extiende el proyecto pero no debería ser una prioridad.

ID	CU-1
Nombre	Iniciar el videojuego
Descripción	El usuario ejecuta el programa.
Prioridad	Vital

ID	CU-2
Nombre	Cerrar el videojuego
Descripción	El usuario cierra el programa.
Prioridad	Vital

¹en referencia a Caso de Uso

²en referencia al Número asociado al mismo

ID	CU-3
Nombre	Seleccionar y Ejecutar opción
Descripción	El usuario escoge una de las opciones posibles para ser ejecutada.
Prioridad	Vital

ID	CU-4
Nombre	Ejecutar Agente contra Agente
Descripción	El usuario escoge una de las opciones en las que el agente compite contra otra versión de sí mismo.
Prioridad	Vital

ID	CU-5
Nombre	Ejecutar Agente contra Agente con visualización
Descripción	El usuario escoge la opción de visualizar un combate entre un agente y otra versión de sí mismo.
Prioridad	Vital

ID	CU-6
Nombre	Ejecutar Agente contra Agente sin visualización
Descripción	El usuario escoge la opción de simular un combate entre un agente y otra versión de sí mismo sin realizar una visualización del mismo.
Prioridad	Vital

ID	CU-7
Nombre	Ejecutar Jugador contra Agente
Descripción	El usuario escoge la opción que le permite competir contra el agente.
Prioridad	Vital

ID	CU-8
Nombre	Ejecutar Jugador contra Jugador
Descripción	El usuario escoge la opción que le permite competir contra otro usuario.
Prioridad	Importante

ID	CU-9
Nombre	Visualizar resultados de combates
Descripción	El usuario visualiza los resultados acumulados de los combates que han ocurrido en la ejecución del programa.
Prioridad	Deseable

3.1.3. Especificación de requisitos

Una vez definidos los casos de uso se puede proceder a realizar un análisis detenido de cada uno de ellos para así extraer los requisitos que se deben cumplir para hacer posible su ejecución. Esta sección estará dedicada a clasificar y describir con el menor grado de ambigüedad posibles cada uno de los requisitos del proyecto. Para este fin utilizaremos el estándar IEEE830 [4], en el que se define la diferenciación entre tres tipos de requisitos:

- **Requisitos de información:** Identificados con códigos de la forma RI-*N*
- **Requisitos funcionales:** Identificados con códigos de la forma RF-*N*
- **Requisitos no funcionales:** Identificados con códigos de la forma RNF-*N*

Algunos de los campos que se utilizarán para describir los requisitos son comunes, tales como el identificador (ID), nombre, descripción y criticidad. En aras de mantener un buen nivel de consistencia se utilizará una escala similar a la de la importancia de los casos de uso para definir la criticidad, pudiendo tomar la misma los siguientes valores:

- **Vital:** Es fundamental para completar el proyecto.
- **Importante:** Aumenta de forma significativa la calidad y funcionamiento del proyecto pero no tiene la cualidad de indispensable.
- **Deseable:** Extiende el proyecto pero no debería ser una prioridad.

3.1.3.1. Requisitos de información

Este subapartado contendrá los requisitos de información del proyecto. Este tipo de requisito especifica los datos que deberán ser almacenados por el sistema para permitir así el cumplimiento de otros requisitos ya sea de forma parcial o total. A continuación se muestran formalmente los requisitos de información del proyecto:

ID	RI-1
Nombre	Datos aprendidos por el agente
Requisitos asociados	<ul style="list-style-type: none">■ RF-5: Salir de la aplicación■ RF-7: Moverse en el área de combate■ RF-8: Atacar al enemigo■ RF-9: Defenderse del enemigo■ RF-13: Visualizar combate entre agentes■ RF-14: Simular múltiples combates
Descripción	El sistema deberá almacenar los datos que representan el conocimiento aprendido por el agente, sea cual sea su forma.
Datos específicos	<ul style="list-style-type: none">■ Estados visitados previamente por el agente■ Acciones tomadas por el agente en un determinado estado■ Resultado, positivo o negativo, de las acciones tomadas en un estado determinado.
Criticidad	Vital

ID	RI-2
Nombre	Archivos necesarios para la ejecución del videojuego
Requisitos asociados	Todos los requisitos funcionales están asociados ya que para todas las funcionalidades es necesario tener acceso a estos archivos.
Descripción	El sistema deberá almacenar todos los archivos no relacionados con el código de la aplicación en si misma que se requieren para mostrar los contenidos de la aplicación
Datos específicos	<ul style="list-style-type: none"> ■ Imágenes que contienen las animaciones de los personajes. ■ Archivos de audio que contienen los diferentes sonidos del videojuego. ■ Archivos que contienen las diferentes fuentes usadas para mostrar el texto por pantalla. ■ Archivos que contienen los "Shaders"³ utilizados para los diferentes efectos dentro del videojuego.
Criticidad	Vital

3.1.3.2. Requisitos de funcionales

Los requisitos funcionales son utilizados para definir las funciones que un sistema debe ser capaz de realizar, definiendo de esta forma el comportamiento que tendrá el software a nivel interno. La especificación de los mismos muestra como, a partir de su cumplimiento, se llevarán a la práctica los casos de uso definidos para el proyecto.

ID	RF-1
Nombre	Visualizar el menú
Requisitos asociados	<ul style="list-style-type: none">■ RF-2: Moverse en el menú■ RF-3: Ejecutar una opción del menú■ RF-12: Volver al menú
Descripción	El sistema deberá mostrar un menú con las diferentes opciones de combate que es posible ejecutar en la aplicación.
Precondición	Ninguna
Secuencia normal	<ol style="list-style-type: none">1. El usuario ejecuta la aplicación.2. El usuario visualiza las opciones disponibles distinguiendo la preseleccionada por defecto.
Secuencia alternativa 1	<ol style="list-style-type: none">1. El vuelve al menú después de un combate.2. El usuario visualiza las opciones disponibles distinguiendo la preseleccionada por defecto.
Postcondición	Se pueden ver y distinguir todas las opciones del menú y seleccionar una de ellas.
Criticidad	Vital
Criterio de validación	Se considerará este requisito como aceptado cuando el usuario pueda visualizar correctamente el menú sea cual sea la forma en la que se llegue a él (primera ejecución o después de combate).

ID	RF-2
Nombre	Moverse en el menú
Requisitos asociados	<ul style="list-style-type: none"> ■ RF-1: Visualizar el menú ■ RF-3: Ejecutar una opción del menú ■ RF-12: Volver al menú
Descripción	El sistema debe permitir viajar entre las diferentes opciones del menú mostrando claramente que opción es la seleccionada actualmente.
Precondición	Se esta visualizando el menú de la aplicación.
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario pulsa el botón para moverse en el menú (hacia arriba o abajo).
Postcondición	Se cambia la opción seleccionada a la siguiente o anterior dependiendo del botón pulsado, abajo o arriba respectivamente.
Criticidad	Vital
Criterio de validación	Se considerará este requisito como aceptado cuando el usuario pueda cambiar de opción en el menú utilizando los botones dedicados a ello y se visualice dicho cambio de forma inequívoca.

ID	RF-3
Nombre	Ejecutar una opción del menú
Requisitos asociados	<ul style="list-style-type: none">■ RF-1: Visualizar el menú■ RF-2: Moverse en el menú■ RF-12: Volver al menú
Descripción	El sistema deberá permitir la ejecución de la opción actualmente seleccionada en el menú de la aplicación.
Precondición	Se está visualizando el menú de la aplicación y hay una opción seleccionada.
Secuencia normal	<ol style="list-style-type: none">1. El usuario pulsa el botón de ejecutar la opción seleccionada del menú.
Postcondición	Se ejecuta la acción seleccionada y se pasa a la escena asociada a dicha acción.
Criticidad	Vital
Criterio de validación	Se considerará este requisito como aceptado cuando el usuario pueda ejecutar todas las opciones del menú y se pase a la escena que se ha asociado con cada una de ellas.

ID	RF-4
Nombre	Alternar apertura de la consola con resultados
Requisitos asociados	<ul style="list-style-type: none"> ■ RF-4: Alternar apertura de la consola con resultados
Descripción	El sistema deberá permitir alternar entre la apertura y cierre de una consola donde se muestren los resultados de los combates previos realizados en esa ejecución del programa.
Precondición	Se han ejecutado combates en la presente ejecución del programa.
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario pulsa el botón de abrir/cerrar la consola.
Postcondición	La consola cambia al estado de visualización distinto al que estaba, se cierra si estaba abierta y se abre si estaba cerrada.
Criticidad	Importante
Criterio de validación	Se considerará este requisito como aceptado si la apertura y cierre de la consola de resultados es funcional en cualquier estado posible en el que el usuario pueda visualizar la aplicación.

ID	RF-5
Nombre	Salir de la aplicación
Descripción	El sistema deberá permitir salir de la aplicación sea cual sea el estado en el que se encuentre y sin producir errores ni en la presente ni en futuras ejecuciones.
Precondición	La aplicación está en ejecución en cualquier estado.
Secuencia normal	<ol style="list-style-type: none">1. El usuario ejecuta la opción del menú dedicada a cerrar la aplicación.
Secuencia alternativa 1	<ol style="list-style-type: none">1. El usuario cierra la ventana de la aplicación haciendo uso de las funcionalidades del sistema de ventanas.
Postcondición	Se ha terminado la ejecución de la aplicación.
Criticidad	Vital
Criterio de validación	Se considerará este requisito como aceptado si el cierre de la aplicación no produce errores sea cual sea el estado actual del programa y usando una de las dos secuencias explicadas.

ID	RF-6
Nombre	Entrar en la escena del combate
Requisitos asociados	<ul style="list-style-type: none"> ■ RF-3: Ejecutar una opción del menú
Descripción	El sistema deberá permitir cambiar a una escena de combate si así lo requiere la ejecución de una de las opciones del menú. Al realizar esta acción se visualizará el combate y se podrá interactuar con el enemigo.
Precondición	Se ha seleccionado una de las opciones del menú que requiere pasar a la escena del combate.
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario pulsa el botón de entrar en la escena del combate.
Postcondición	La aplicación se encuentra ahora en una escena de combate.
Criticidad	Vital
Criterio de validación	Se considerará este requisito como aceptado si se entra en una escena de combate al pulsar una opción del menú que lo requiera.

ID	RF-7
Nombre	Moverse en el área de combate.
Requisitos asociados	<ul style="list-style-type: none">■ RF-8: Atacar al enemigo■ RF-9: Defenderse del enemigo
Descripción	El sistema deberá permitir que el usuario mueva a su personaje en el contexto del combate.
Precondición	La aplicación se encuentra en una escena de combate.
Secuencia normal	<ol style="list-style-type: none">1. El usuario mueve la palanca o "joystick" en la dirección que quiere mover su personaje.
Postcondición	El personaje se mueve en la dirección indicada por el usuario.
Criticidad	Vital
Criterio de validación	Se considerará este requisito como aceptado si el personaje se mueve hacia la dirección indicada por el usuario.

ID	RF-8
Nombre	Atacar al enemigo
Requisitos asociados	<ul style="list-style-type: none">■ RF-7: Moverse en el área de combate■ RF-9: Defenderse del enemigo
Descripción	El sistema deberá permitir que un personaje dañe a otro mediante la utilización de un ataque.
Precondición	La aplicación se encuentra en una escena de combate.
Secuencia normal	<ol style="list-style-type: none">1. El usuario pulsa el botón de atacar.
Secuencia alternativa 1	<ol style="list-style-type: none">1. El agente decide realizar la acción de atacar.
Postcondición	El personaje que debe realizar la acción de atacar comienza la ejecución del ataque.
Criticidad	Vital
Criterio de validación	Se considerará este requisito como aceptado si tanto el usuario como el agente son capaces de atacar cuando deciden realizar esa acción dentro de la escena de combate.

ID	RF-9
Nombre	Defenderse del enemigo
Requisitos asociados	<ul style="list-style-type: none">■ RF-7: Moverse en el área de combate■ RF-8: Atacar al enemigo
Descripción	El sistema deberá permitir que un usuario realice una acción defensiva contra su contrincante de forma que no pueda recibir daño.
Precondición	La aplicación se encuentra en una escena de combate.
Secuencia normal	<ol style="list-style-type: none">1. El usuario pulsa el botón de defenderse.
Secuencia alternativa 1	<ol style="list-style-type: none">1. El agente decide realizar la acción de defenderse.
Postcondición	El personaje que debe realizar la acción de defenderse se encuentra en un estado en el que no puede recibir daño.
Criticidad	Vital
Criterio de validación	Se considerará este requisito como aceptado si el personaje se encuentra en un estado de defensa cuando deciden realizar esta acción dentro de la escena de combate.

ID	RF-10
Nombre	Ganar/Perder partida
Requisitos asociados	<ul style="list-style-type: none"> ■ RF-11: Agotar el tiempo de combate
Descripción	El sistema deberá permitir que uno de los personajes gane la partida si la vida de su contrincante llega a cero. Haciendo de forma efectiva que uno de los personajes pierda y otro gane.
Precondición	La aplicación se encuentra una escena de combate.
Secuencia normal	<ol style="list-style-type: none"> 1. Uno de los personajes lleva a cabo una acción que desemboca en que la vida de uno de ellos llegue a cero.
Postcondición	Se muestra por pantalla claramente cual de los personajes ha ganado y se detiene el combate.
Criticidad	Vital
Criterio de validación	Se considerará este requisito como aceptado si siempre que la vida de uno de los personajes llegue a cero se detiene la pelea y se nombra ganador de la misma al personaje superviviente.

ID	RF-11
Nombre	Agotar el tiempo de combate
Requisitos asociados	<ul style="list-style-type: none">■ RF-10: Ganar/Perder partida
Descripción	La aplicación deberá permitir que un combate termine sin que la vida de uno de los personajes llegue a cero si se termina el tiempo máximo establecido.
Precondición	La aplicación se encuentra en una escena de combate.
Secuencia normal	<ol style="list-style-type: none">1. El usuario espera a que el tiempo de combate llegue a cero.
Postcondición	Se detiene el combate.
Criticidad	Importante
Criterio de validación	Se considerará este requisito como aceptado si sea cual sea el estado de un combate este se detiene al llegar el tiempo disponible para el mismo a cero.

ID	RF-12
Nombre	Volver al menú
Requisitos asociados	<ul style="list-style-type: none">■ RF-1: Visualizar el menú
Descripción	El sistema deberá permitir volver al menú principal desde la escena de combate.
Precondición	La aplicación se encuentra en una escena de combate.
Secuencia normal	<ol style="list-style-type: none">1. El usuario pulsa el botón de volver al menú.
Postcondición	Se detiene la pelea y se vuelve al menú de la aplicación.
Criticidad	Importante
Criterio de validación	Se considerará este requisito como aceptado si siempre que la aplicación se encuentre mostrando una escena de combate es posible volver al menú principal utilizando el botón dedicado para ello.

ID	RF-13
Nombre	Visualizar combate entre agentes
Requisitos asociados	<ul style="list-style-type: none">■ RF-6: Entrar en la escena de combate■ RF-10: Ganar/Perder partida■ RF-11: Agotar el tiempo de combate
Descripción	El sistema deberá permitir visualizar la totalidad de un combate entre el agente y otra versión de si mismo sin que el usuario tenga que controlar a ningún personaje dentro de la escena de combate.
Precondición	Se ejecuta la opción del menú dedicada a visualizar el combate entre dos agentes.
Secuencia normal	<ol style="list-style-type: none">1. El usuario pulsa el botón dedicado a comenzar la pelea entre dos agentes.
Postcondición	Se entra en la escena de combate con los dos personajes siendo controlados por un agente.
Criticidad	Vital
Criterio de validación	Se considerará este requisito como aceptado si se puede visualizar un combate completo entre un agente y otra versión de si mismo de la misma forma que se visualizaría un combate entre regular entre el usuario y un agente o entre dos usuarios.

ID	RF-14
Nombre	Simular múltiples combates
Requisitos asociados	<ul style="list-style-type: none">■ RF-6: Entrar en la escena de combate
Descripción	El sistema deberá permitir que se realicen varios combates entre dos versiones del agente de forma acelerada para permitir que el mismo complete el proceso de aprendizaje en un tiempo razonable.
Precondición	Se ejecuta la opción del menú dedicada a simular varios combates entre dos agentes.
Secuencia normal	<ol style="list-style-type: none">1. El usuario pulsa el botón dedicado a simular varios combates entre agentes.
Postcondición	Se comienza la simulación de los combates.
Criticidad	Vital
Criterio de validación	Se considerará este requisito como aceptado si se pueden realizar varias simulaciones de forma iterativa en las que el agente visite los diferentes estados.

ID	RF-15
Nombre	Comandos en la consola
Requisitos asociados	<ul style="list-style-type: none">■ RF-4: Alternar apertura de la consola con resultados
Descripción	El sistema deberá permitir ejecutar comandos utilizando la consola para realizar acciones específicas dentro de la aplicación de forma aislada.
Precondición	La consola de la aplicación está siendo visualizada.
Secuencia normal	<ol style="list-style-type: none">1. El usuario introduce un comando en formato de texto en la consola.2. El usuario pulsa el botón de ejecutar el comando.3. El usuario visualiza el resultado.
Excepciones secuencia normal	<ol style="list-style-type: none">4. El usuario no percibe ningún resultado porque el comando no existe.
Postcondición	El comando ha sido procesado y ejecutado.
Criticidad	Deseable
Criterio de validación	Se considerará este requisito como aceptado si se puede ejecutar un comando en la consola de la aplicación de forma que si el mismo es correcto se ejecute y si no lo es no ocurra nada.

3.1.3.3. Requisitos no funcionales

Los requisitos no funcionales especifican necesidades del sistema referentes a su operación, esto puede referirse a rendimiento, tiempos de respuesta, capacidad, etc. Al contrario que los requisitos funcionales, los requisitos no funcionales se centran en las características de funcionamiento de esas funcionalidades y no en las funcionalidades en si mismas.

En este apartado se definirán los requisitos no funcionales de este proyecto formalmente:

ID	RNF-1
Nombre	Rendimiento de la aplicación
Descripción	La aplicación, especialmente en las escenas de combate, deberá visualizarse a 60 fotogramas por segundo para garantizar la fluidez de cara al usuario.
Criticidad	Importante
Criterio de validación	Se considerará este requisito como cumplido si durante un número suficiente de combates consecutivos, los fotogramas por segundo medios de la aplicación no caen por debajo de 60.

ID	RNF-2
Nombre	Velocidad de las simulaciones
Descripción	En aras de permitir la realización del proceso de aprendizaje del agente en un tiempo razonable se deberá permitir la simulación de combates con el tiempo interno del juego sustancialmente acelerado.
Criticidad	Vital
Criterio de validación	Se considerará este requisito como cumplido si el agente es capaz de competir con una de las implementaciones basadas en reglas y ganar más de un 50 % de las veces después de menos de 15 minutos de entrenamiento.

ID	RNF-3
Nombre	Extensibilidad del motor
Descripción	La aplicación deberá de estar correctamente separada en subsistemas independientes conectados mediante un bus de mensajes que permita agregar nuevos subsistemas sin afectar a los existentes.
Criticidad	Importante
Criterio de validación	Se considerará este requisito como cumplido si se sigue el patrón de arquitectura de bus de mensajes y se puede agregar un nuevo subsistema sin hacer ninguna modificación a los existentes.

ID	RNF-4
Nombre	Facilidad para depurar
Descripción	Se debe permitir ejecutar la ejecución de comandos en tiempo de ejecución para facilitar la depuración de subsistemas concretos.
Criticidad	Deseable
Criterio de validación	Se considerará este requisito como cumplido si se pueden insertar mensajes en el bus de mensajes que conecta los subsistemas que componen el programa desde la consola de la aplicación.

ID	RNF-5
Nombre	Aplicación autocontenida
Descripción	La aplicación deberá de poder ser distribuida de forma que no se requiera la instalación de frameworks, librerías externas u otro archivos para su ejecución.
Criticidad	Importante
Criterio de validación	Se considerará este requisito como cumplido si se puede ejecutar la aplicación sin realizar ningún cambio sobre una instalación limpia de los sistemas operativos para los que se ha construido. Además todos los archivos que requiera la aplicación deberán estar dentro del ejecutable, siendo su existencia transparente al usuario.

ID	RNF-6
Nombre	Extensibilidad en términos de escenas
Descripción	La aplicación se deberá de componer por escenas de forma que siempre haya solamente una escena activa, pudiendo cambiar entre diferentes escenas con facilidad y agregar nuevas escenas si es necesario.
Criticidad	Importante
Criterio de validación	Se considerará este requisito como cumplido si se pueden agregar escenas a la aplicación y cambiar entre ellas sin hacer ninguna modificación sobre el código que existía previamente tanto en términos de escenas como del propio motor.

ID	RNF-7
Nombre	Documentación
Descripción	La documentación generada para el proyecto debe priorizar la claridad y simplicidad cuando sea posible. Teniendo en cuenta los conocimientos de los lectores para los que está orientada.
Criticidad	Importante
Criterio de validación	Se considerará este requisito como cumplido si la documentación carece de apartados o secciones que no aportan contenido relevante o son redundantes. Además, un lector externo tiene que ser capaz de comprender los conceptos empleados sin necesidad de una amplia experiencia previa en la materia.

ID	RNF-8
Nombre	Usabilidad de la interfaz
Descripción	La interfaz gráfica de la aplicación deberá estar diseñada para que sea fácilmente usable por cualquier tipo de usuario con cierto conocimiento de videojuegos de un estilo similar. Siguiendo las heurísticas definidas por Nielsen [5] siempre que sea posible.
Criticidad	Importante
Criterio de validación	Se considerará este requisito como cumplido si al menos 3 de 5 usuarios sin experiencia lo evalúan como usable al terminar pruebas supervisadas de la aplicación.

3.1.3.4. Matriz de trazabilidad

Como parte final de análisis de requisitos se muestra una matriz de trazabilidad en la Tabla 3.1 que hará corresponder a los requisitos funcionales definidos anteriormente con los casos de uso a los que están asociados

Cuadro 3.1: Matriz de trazabilidad de requisitos contra casos de uso

	CU-1	CU-2	CU-3	CU-4	CU-5	CU-6	CU-7	CU-8	CU-9
RF-1	✓		✓						
RF-2			✓						
RF-3			✓	✓					
RF-4			✓						✓
RF-5		✓							
RF-6			✓	✓			✓	✓	
RF-7							✓	✓	
RF-8							✓	✓	
RF-9							✓	✓	
RF-10			✓				✓	✓	✓
RF-11			✓				✓	✓	
RF-12			✓	✓					
RF-13				✓	✓				
RF-14				✓		✓			
RF-15			✓						

Capítulo 4

Gestión del proyecto

Dentro de la ingeniería de software, una de las partes esenciales para la realización de proyectos considerados exitosos es la gestión de proyectos. La realización de una buena gestión no se puede considerar, ni mucho menos, una garantía de que el proyecto a gestionar vaya a resultar un éxito. Sin embargo, si elegimos ignorar o realizar una mala gestión de nuestro proyecto sí podremos considerar que nos encontraremos en una situación proclive para un proyecto fallido. Durante toda la extensión del ciclo de vida de nuestro proyecto se utilizará la gestión de proyectos como un método que nos ayudará a lograr la obtención de un producto final ajustado a todas las necesidades y restricciones presentes, sea cual sea la índole de las mismas (tiempo, costes, requisitos, etc.).

Dedicaremos este capítulo a definir y explicar la gestión de nuestro proyecto en todas sus partes. Se determinará y explicará el análisis de riesgos, la metodología de desarrollo empleada, la gestión de configuración, la planificación temporal y la estimación de costes.

4.1. Gestión de riesgos

Citando al PMBOK [1] la definición de riesgo es:

“[...]un evento o condición incierta que, de producirse, tiene un efecto positivo o negativo en uno o más de los objetivos del proyecto, tales como el alcance, el cronograma, el costo y la calidad.”

Las causas de un riesgo pueden ser varias y diversas y, si este finalmente se diera, los impactos que puede producir también pueden ser numerosos. Dentro de las causas, su índole puede ser de tipos muy variados, desde un requisito mal especificado a una restricción que no existe pasando por supuestos que no se

ajustan a realidad o fallos en procesos externos al proyecto. Si alguna de estas situaciones se produce podría haber un impacto relevante sobre los objetivos definidos previamente para el proyecto como lo serían el alcance, el coste, el cronograma o la calidad.

El impacto suele ser generalmente negativo aunque ocasionalmente puede ser beneficioso para el proyecto. En el proyecto al que se refiere esta memoria realizaremos una gestión de riesgos enfocada a los que tienen una naturaleza negativa y que por lo tanto podrían afectar a los objetivos de forma perjudicial. Una vez dicho esto, hay que considerar las diferentes estrategias que se pueden utilizar para abordar las amenazas o riesgos con impacto negativo en caso de materializarse y serán las siguientes:

- **Evitar:** Siguiendo esta estrategia de respuesta el equipo del proyecto intentará eliminar la amenaza o proteger al proyecto del posible impacto de forma preventiva. Comúnmente implica modificar la planificación para evitar completamente la amenaza.
- **Transferir:** Al transferir un riesgo el equipo traslada el impacto asociado al mismo a un tercero, librándose así de la responsabilidad de dar una respuesta si el riesgo se da. Suele incluir el pago de una prima de riesgo a la entidad que asumirá el impacto del mismo.
- **Mitigar:** Mitigar un riesgo implica actuar para reducir o bien la probabilidad de un riesgo o bien el impacto que este tendría en el proyecto, llevando uno de estos aspectos o ambos a un umbral aceptable. Habitualmente es más efectivo realizar acciones preventivas que intentar lidiar con el riesgo una vez que ha ocurrido.
- **Aceptar:** Si el equipo decide reconocer el riesgo y no hacer nada al respecto a menos que este se materialice la estrategia que se está utilizando es la de aceptar dicho riesgo. Esta aproximación se suele utilizar si no es posible o rentable utilizar alguna de las otras estrategias. Esto puede implicar no realizar ninguna acción si el riesgo ocurra (estrategia pasiva) o establecer una reserva para contingencias que pueda aportar los recursos, tiempo o dinero necesarios para gestionar el riesgo (estrategia activa).

Dada la índole del proyecto y considerando el hecho de que solo hay un trabajador encargado del mismo se intentará evitar la transferencia de un riesgo al ser esta estrategia propensa a generar costes adicionales. Por razones similares, la aceptación de un riesgo se deberá considerar única y exclusivamente si ninguna de las otras estrategias es aplicable.

Cuadro 4.1: Valores posibles de probabilidad e impacto

Calificativo	Probabilidad	Impacto
Muy Bajo	$p < 0,10$	$i < 0,10$
Bajo	$0,10 \leq p < 0,30$	$0,10 \leq i < 0,20$
Moderado	$0,30 \leq p < 0,50$	$0,20 \leq i < 0,40$
Alto	$0,50 \leq p < 0,70$	$0,40 \leq i < 0,80$
Muy Alto	$0,70 \leq p$	$0,80 \leq i$

Para realizar un buen análisis de riesgos se debe comenzar por una identificación de la mayor cantidad de riesgos posible dentro de unos límites lógicos para luego analizar la importancia de cada uno teniendo en cuenta su probabilidad e impacto. Finalmente se deberán establecer estrategias a seguir en caso de que los riesgos se den, priorizando los más importantes en caso de que no sea posible gestionarlos todos.

4.1.1. Especificación de riesgos

Se dedicará este apartado a mostrar las especificaciones formales de los riesgos que contendrán su descripción, probabilidad, impacto y los planes diseñados para cada uno de ellos. La probabilidad¹ e impacto² podrán tomar los valores definidos en el Cuadro 4.1. El identificador utilizado para los riesgos tendrá la forma RSK^3-N

¹indicada como p dentro de la tabla

²indicada como i dentro de la tabla

³En referencia a *risk*, riesgo en inglés

ID	RSK-1
Nombre	Retraso en la planificación
Descripción	Se debe contemplar la posibilidad de que ocurran retrasos en la planificación dada la inexperiencia del desarrollador a la hora de trabajar en un proyecto de esta envergadura y con algunas de las tecnologías a utilizar.
Probabilidad de ocurrencia	Alta
Impacto	Muy Alto
Plan de Prevención	Mitigar: Realizar reuniones periódicas con los tutores del proyecto para resolver dudas y ver el avance del mismo.
Plan de Contingencia	Aceptar: Redefinir el alcance del proyecto volviendo a evaluar los requisitos para priorizar los mas necesarios dejando los no vitales fuera del proyecto.

ID	RSK-2
Nombre	Cambio en los requisitos
Descripción	Es posible que una vez avanzado el proyecto se considere que algunos de requisitos necesitan ser añadidos, modificados o eliminados, especialmente dado que se dedicará cierta parte del proyecto a investigar la utilidad de técnicas de inteligencia artificial sobre las cuales se desconoce si serán de utilidad.
Probabilidad de ocurrencia	Alta
Impacto	Alto
Plan de Prevención	Mitigar: Realizar un diseño de la aplicación de la forma más modular posible para evitar grandes refactorizaciones a la hora de hacer modificaciones importantes.
Plan de Prevención 2	Mitigar: Utilizar una metodología de trabajo ágil que sea capaz de aceptar y soportar cambios sustanciales en los requisitos del proyecto.

ID	RSK-3
Nombre	Falta de comprensión de conceptos
Descripción	Puede ocurrir que el desarrollador no esté tan familiarizado con los conceptos relacionados con inteligencia artificial como es necesario para la realización del proyecto, dificultando así la implementación correcta de cualquiera de sus técnicas.
Probabilidad de ocurrencia	Media
Impacto	Alto
Plan de Prevención	Mitigar: Poner a disposición del desarrollador bibliografía que explique los conceptos que pueden ser necesarios.
Plan de Contingencia	Aceptar: Dedicar las reuniones necesarias con los tutores a explicar los conceptos al desarrollador hasta que el nivel de comprensión de los mismos sea suficiente.

ID	RSK-4
Nombre	Baja del desarrollador del proyecto
Descripción	Se puede dar el caso de que, por diversas causas como problemas de salud, económicos o de otra índole, el desarrollador del proyecto se vea obligado a dejar de trabajar en él.
Probabilidad de ocurrencia	Baja
Impacto	Muy Alto
Plan de Contingencia	Aceptar: Aplazar el proyecto hasta que el desarrollador se encuentre en condiciones de volver a trabajar en él.

ID	RSK-5
Nombre	Baja de los tutores
Descripción	Se puede dar el caso de que, por diversas causas como problemas de salud, económicos o de otra índole, uno o ambos tutores del proyecto se vean obligados a dejar de trabajar en él.
Probabilidad de ocurrencia	Bajo
Impacto	Alto
Plan de Contingencia	Aceptar: En caso de que solo uno de los tutores deje de poder trabajar en el proyecto se continuará con el desarrollo con la ayuda del otro tutor. Si se da el caso de que ambos tienen que dejar de trabajar se deberá buscar un tutor o grupo de tutores como sustitutos.

ID	RSK-6
Nombre	Fallo en el equipo de trabajo
Descripción	Puede ocurrir que el equipo utilizado para el desarrollo quede inutilizado por razones de software o hardware lo que impide continuar con el proyecto en dicho sistema.
Probabilidad de ocurrencia	Baja
Impacto	Alto
Plan de Contingencia	Aceptar: Se podrá hacer uso de equipos de la universidad para la realización del proyecto, si estos no son compatibles con alguna de las tecnologías usadas se necesitará adquirir un equipo sustituto cuya amortización tendrá que ser considerada dentro de los costes del proyecto.

ID	RSK-7
Nombre	Perdida del proyecto y su documentación
Descripción	Puede ocurrir que un fallo en el equipo de trabajo o en su almacenamiento haga inaccesible o elimine el topo o parte del trabajo realizado para el proyecto y/o para su documentación.
Probabilidad de ocurrencia	Baja
Impacto	Muy Alto
Plan de Prevención	Evitar: Utilizar un sistema de control de versiones en la nube que permita guardar no solo todo el código y archivos necesarios para la ejecución del programa sino también todos los documentos que generan la documentación pudiendo recuperarlos desde cualquier otro equipo.

ID	RSK-8
Nombre	Rendimiento insuficiente del videojuego
Descripción	Puede ocurrir que, dada la inexperiencia como programador del desarrollador principal, el videojuego no cumpla los estándares de rendimiento esperados del mismo y no brinde una experiencia jugable.
Probabilidad de ocurrencia	Baja
Impacto	Muy Alto
Plan de Prevención	Mitigar: Dedicar un tiempo en la fase de análisis y diseño a familiarizarse con las tecnologías y técnicas utilizadas en la programación de videojuegos cuando el rendimiento es un factor importante.
Plan de Contingencia	Aceptar: Volver a realizar la etapa de diseño e implementación de forma que se eliminen las partes menos indispensables y más costosas del videojuego en aras de buscar un rendimiento superior.

ID	RSK-9
Nombre	Imposibilidad de realizar simulaciones aceleradas
Descripción	Dado que la realización de simulaciones dentro del entorno de un videojuego en tiempo real no es una práctica común es posible que las tecnologías escogidas no permitan realizar dicho proceso lo suficientemente rápido como para que el agente aprenda en un tiempo razonable.
Probabilidad de ocurrencia	Moderada
Impacto	Muy Alto
Plan de Prevención	Mitigar: Realizar un prototipo inicial con la tecnología escogida para la implementación del videojuego lo más temprano posible en el proyecto con el fin de averiguar sus capacidades a la hora de acelerar las simulaciones. En este momento se considerará si es necesario cambiar la tecnología y/o realizar una implementación más eficiente.
Plan de Contingencia	Aceptar: Cambiar la tecnología y/o realizar cambios en el diseño e implementación buscando un aumento en la eficiencia que posibilite realizar simulaciones a una velocidad suficiente.

ID	RSK-10
Nombre	Agente sin capacidad de aprendizaje
Descripción	Es posible que dada la inexperiencia del desarrollador en términos de implementación de técnicas de inteligencia artificial el funcionamiento no sea el correcto y el agente no sea capaz de aprender y actuar dentro del entorno del juego.
Probabilidad de ocurrencia	Baja
Impacto	Alto
Plan de Prevención	Mitigar: Dedicar parte del tiempo en las reuniones con los tutores a comprobar el correcto funcionamiento de las técnicas implementadas.
Plan de Contingencia	Aceptar: Cuando los errores en la implementación eviten el correcto funcionamiento del agente se dedicará una reunión específica a depurar dicha implementación y solucionar los errores presentes.

4.1.2. Control de riesgos

Controlar los riesgos implica implementar los planes de respuesta incluidos en la sección de especificación así como dar seguimiento a los riesgos identificados, monitorizarlos, identificar riesgos nuevos y en general evaluar el proceso de gestión de riesgos que se está realizando. A términos de este proyecto se considerará principalmente en el proceso de ejecutar las respuestas a los riesgos planificados que se han incluido en el apartado de especificación ya que, dado el tamaño y tiempo que se debe dedicar al proyecto, se podría correr el riesgo de dedicar demasiado tiempo a procesos de gestión de riesgos comparado con el tiempo dedicado al proyecto en si mismo.

4.1.2.1. Acciones correctivas aplicadas

Este apartado incluirá todo tipo de acción que se ha llevado a cabo para gestionar un riesgo que se ha materializado. En este sentido se especificarán las

circunstancias en las que se ha dado el riesgo, la probabilidad e impacto una vez gestionado el riesgo y aplicado el plan apropiado, una descripción de las acciones realizadas y el efecto que las mismas han tenido en el proyecto.

Cuadro 4.2: Acción correctiva sobre la imposibilidad de realizar simulaciones aceleradas

ID	AC-1
ID del riesgo	RSK-9: Imposibilidad de realizar simulaciones aceleradas
Nombre	Agente sin capacidad de aprendizaje
Descripción	<p>Al llevar a cabo el plan de prevención definido para este riesgo se descubrió que Unity, el motor elegido sobre el cual se desarrollaría la aplicación, no era capaz de escalar el tiempo de forma significativa sin que mecánicas básicas dentro del mismo dejaran de funcionar.</p> <p>El motor ofrece la funcionalidad de escalar el tiempo mediante el uso de la variable <i>Time.timeScale</i> pero al intentar multiplicar la velocidad del tiempo interno del juego por valores relativamente pequeños como 20 o 30 las físicas del motor dejaban de funcionar correctamente. Para poner en contexto estos valores, en la implementación final realizada por el desarrollador el tiempo se escala por aproximadamente 600 veces sin que se modifique su funcionamiento en ningún sentido.</p>

Acción correctiva	<p>La acción correctiva coincide con el plan de contingencia definido en la especificación del riesgo. Esto implicaba considerar una reimplentación de la aplicación ya fuera con la misma u otra tecnología. Dado que el origen del problema parecía estar relacionado con el motor utilizado y no con la implementación en si misma se decidió buscar tecnologías alternativas.</p> <p>Al considerar otros motores se observó que este tipo de funcionalidades era aún mas inusual de lo que en un principio parecía y había muchas posibilidades de que ninguno fuera capaz de cumplir nuestras necesidades.</p> <p>Como no se podía correr el riesgo de utilizar tiempo en preparar otro prototipo y que no funcionara otra vez se decidió hacer una implementación del videojuego con C++ y el uso de librerías gráficas de forma que fuera posible desactivar la visualización de los combates y así conseguir que todo el código que se ejecuta en las simulaciones fuera realizado por nosotros, dándonos un nivel de control superior y permitiendo realizar las optimizaciones pertinentes.</p>
--------------------------	---

Efecto en el proyecto	<p>El impacto sobre el proyecto ha sido significativo pues la acción correctiva implica reconsiderar el alcance de algunos de los objetivos, principalmente los relacionados con la implementación de diversas técnicas de inteligencia artificial para comprobar su efectividad.</p> <p>Esto es debido a que el tiempo dedicado a implementar una aplicación de esta índole con un motor ampliamente usado por la comunidad y que aporta muchas facilidades es notablemente menor que el necesario para implementar en un lenguaje de programación como C++ un motor simple desde cero y agregar al mismo las funcionalidades necesarias.</p> <p>El plan de contingencia aplicado ha implicado no solo reconsiderar los requisitos hasta obtener los que se muestran en este documento, sino modificar la planificación temporal y agregar tareas relacionadas con el aprendizaje, diseño e implementación de la nueva aplicación como se muestra en los apartados correspondientes.</p>
Nueva probabilidad de ocurrencia	Baja
Nuevo impacto	Alto

4.2. Gestión de la configuración

INTECO[6] define la gestión de configuración como el proceso:

[...]cuyo propósito es establecer y mantener la integridad de los productos de trabajo.

Además, gracias a su guía orientada a la gestión de la configuración, se pueden obtener una serie de procesos orientados a lograr proteger esa integridad de los productos de trabajo de este proyecto. Dichos procesos se resumen en los siguientes:

1. Identificación de los elementos a controlar, que nos servirá como entrada para los procesos siguientes.
2. Definición de procedimientos para controlar los elementos, proceso que se centra en gestionar los cambios sobre la configuración de los diferentes productos.
3. Registro del estado de los elementos, para ser capaz de almacenar el estado actual y pasado de todos los productos.
4. Auditorías de configuración que comprueben si los elementos de configuración cumplen los requisitos necesarios.

En el contexto de un proyecto, la gestión de configuración es muy importante dado que asegura la integridad de los productos desarrollados reduciendo riesgos relacionados con su modificación indeseada, pérdida o entregas erróneas. De esta forma también se eliminan los posibles sobre-esfuerzos generados por errores en la configuración, se controlan los cambios que ha sufrido el proyecto y se realiza una correcta administración de las versiones sobre las cuales se está trabajando.

En el particular caso de este proyecto solo hay un desarrollador activo por lo que todos los procesos que buscan controlar las modificaciones que los trabajadores realizan al proyecto son significativamente simplificados. Por lo tanto, la prioridad de este proceso en nuestro caso será gestionar correctamente las versiones de los productos generados por el proyecto.

Los problemas generados a raíz de no realizar una correcta gestión de configuración, especialmente centrándonos en la gestión de versiones, pueden llegar

a ser muy importantes además de variados. Se puede dar el caso de que se entregue una versión incorrecta del producto, esto puede ser una versión con errores, con cambios no probados o versiones irreproducibles. Además de esto se pueden dar situaciones en las que no se disponga de un inventario completo de los elementos del sistema, que no se puedan recuperar versiones anteriores y, lo que sería más importante aún, que se tenga que realizar el mismo trabajo en múltiples ocasiones por pérdida del mismo.

4.2.1. Herramientas

Dedicaremos este apartado a identificar y describir el uso de las herramientas que se han utilizado dentro del proceso de gestión de la configuración.

- **Google Drive**⁴: Servicio de almacenamiento en la nube que a su vez permite la creación y modificación en tiempo real de archivos de texto, tablas, etc.
- **Dropbox**⁵: Servicio de almacenamiento en la nube capaz de alojar ficheros en la red para que los mismos se sincronicen localmente en todos los equipos con una determinada cuenta.
- **Git y GitHub**⁶: Son, respectivamente el sistema de control de versiones y servicio de repositorios en la red que permiten controlar cambios, gestionar ramas y acceder al proyecto desde cualquier lugar.
- **LaTeX**⁷: Sistema de creación y composición de documentos de alta calidad diseñado para la producción de documentación técnica. Estándar de facto para comunicaciones y publicaciones de carácter científico.

4.2.2. Descripción del soporte de las herramientas al proceso

Comenzando por los dos servicios de almacenamiento en la nube que se han utilizado es posible que surja la duda de porqué utilizar dos herramientas cuya finalidad parece la misma. Pese a que pueda parecer que este es el caso, el uso que se les ha dado a ambos servicios es significativamente distinto.

Google Drive ha sido utilizado para gestionar los documentos para los cuales era importante que tanto el desarrollador como los tutores pudieran tener acceso inmediato, realizar modificaciones en tiempo real y discutir sobre las mismas.

⁴https://www.google.com/intl/es_ALL/drive/

⁵<https://www.dropbox.com/>

⁶<https://github.com/>

⁷<https://www.latex-project.org/>

Este servicio proporciona un completo sistema de comentarios sobre su procesador de texto en línea de forma que los tutores pueden realizar indicaciones sobre partes de texto que necesitan ser revisadas que el desarrollador podrá ver en tiempo real, contestar a los mismos y realizar el cambio. Un buen ejemplo sería el documento de solicitud de aprobación del trabajo ya que el tiempo por iteración del mismo se vio significativamente reducido gracias a este sistema.

Por otra parte, Dropbox ha sido utilizado por la facilidad de sincronización que proporciona. Esto es importante si se necesita trabajar en algún equipo diferente de manera puntual, sea esta propiedad o no del desarrollador. En el caso de que se generara algún archivo en un equipo diferente del dedicado al desarrollo es muy sencillo subirlo a Dropbox ya que estará inmediatamente disponible en el equipo principal. Un buen ejemplo sería la creación de alguno de los archivos que contienen las animaciones del videojuego ya que muy ocasionalmente se realizaban en un equipo distinto. Otro buen uso es el de subir una versión de la aplicación compilada y distribuir un enlace a la misma para que los usuarios elegidos puedan probarla rápidamente.

Hablando ahora de Git y GitHub se podría considerar que la combinación de ambas han constituido la herramienta principal en lo que a gestión de configuración se refiere, especialmente en términos de la gestión de versiones. Mediante su uso se autogeneran versiones con cada *commit* realizado de forma que cada cambio representa automáticamente una nueva versión a la que se puede acceder si es necesario. Además, dado que permite seleccionar el tipo de archivos se deben controlar y cuales no posibilita omitir los archivos pesados como la aplicación compilada y centrarse en el código, documentación y otros archivos que son necesarios para obtener el proyecto. Como nota adicional, también permite agregar lanzamientos o *releases* de forma que se pueden subir ejecutables para tener disponible muy fácilmente versiones probadas y funcionales.

Finalmente, LaTeX ha sido la herramienta elegida para la documentación por su posición como estándar de facto, para facilitar que el desarrollador se familiarice con el uso de la misma y por el gran número de funcionalidades y flexibilidad que proporciona.

4.2.3. Nomenclatura

En aras de conseguir un fácil reconocimiento de las diferentes versiones de la documentación se definirá un esquema de nomenclatura que seguirán los documentos del proyecto. Las características importantes sobre un documento serán su nombre, fecha de creación, versión y tipo por lo que nuestro esquema será el siguiente.

<nombre_del_documento>_<ddmmaa>_v<versión>.<extensión>

Donde <nombre_del_documento> identifica unívocamente el documento, <ddmmaa> se corresponde con la fecha de generación del documento con el formato día, mes, año. <versión> se corresponde con la versión actual del documento y <extensión> indica el formato del mismo.

A modo de ejemplo, si se quiere nombrar esta misma memoria, generada el día 26 de junio de 2017, con la versión 1.1 y con la extensión que indica que se trata de un archivo PDF el nombre a utilizar sería:

MemoriaTrabajoFinDeGrado.260617_v1.1.pdf

Dichos estándares de nomenclatura serán utilizados para las versiones generadas de la documentación y no para otros archivos de código o usados para generar productos del proyecto que ya están gestionados mediante el control de versiones de GitHub.

4.3. Metodología de desarrollo

Como se puede observar fácilmente en el *Chaos Report*[7], un número muy elevado de proyectos son cancelados, abandonados o simplemente terminan habiendo superado los recursos estimados en un principio en términos de coste, tiempo, etc. Una de las razones a las cuales se atribuye este hecho es a una mala o nula elección de la metodología de desarrollo apropiada que se adapte a las necesidades de un proyecto concreto. Si la metodología no está pensada para el tipo de proyecto que se está gestionando se pueden dar problemas que van desde el descontento del cliente hasta la imposibilidad de terminar el proyecto.

Pese a que en proyectos pequeños pueda parecer que no es importante la elección o simple existencia de una metodología, lo cierto es que siempre se está aplicando una u otra metodología aunque el equipo de trabajo no se lo proponga o no se especifique en la documentación. De modo inconsciente se tiende a tener una idea de la estructura que se usará para el proyecto, comúnmente se siguen aproximaciones similares a la metodología en cascada en estas situaciones.

La elección y uso de una metodología concreta debe sugerir una estructuración, planificación y control de los procesos de desarrollo que nos de una base sobre la cual generar toda la gestión del proyecto. Con esto se busca eliminar problemas como altos costes, retrasos en las entregas o calidad baja del producto y documentación mientras se aumenta la calidad y cantidad de información disponible sobre el estado del proyecto.

En un proyecto de las características que nos atañe si es necesario especificar la metodología que se va a usar ya que tiene la envergadura suficiente para ello. Para explicar la elección de la metodología utilizada no se realizará un análisis de las metodologías disponibles para incluir la razón de no haber elegido cada una de ellas. Se explicará como las características del proyecto han desembocado en la elección final de la metodología utilizada.

El equipo de trabajo estará compuesto por un único desarrollador, lo que elimina la necesidad de repartir tareas o gestionar la comunicación dentro del equipo. Además, dada la duración aproximada de poco más de 400 horas se deberá elegir una metodología que no requiera dedicar la mayor parte de tiempo del proyecto a tareas de gestión. Otra característica importante es que se necesita avanzar rápidamente en las etapas iniciales del proyecto dadas las dudas presentes sobre la tecnología (RSK-9) y para permitir así dejar más tiempo para la implementación del agente, etapa que corre el riesgo de necesitar más tiempo de lo estimado.

Los motivos comentados hacen que las metodologías ágiles parezcan la elección propia para el proyecto, especialmente por que permiten acelerar el desarrollo, son más flexibles a cambios, buscan la simplicidad y sus desventajas en términos de organización no son importantes con un equipo de trabajo y duración como la actual. De entre las metodologías ágiles se ha escogido la conocida como **Programación Extrema** o XP.

En esta metodología otorga una flexibilidad al desarrollador que la hace superior para este proyecto dado que aporta libertades al desarrollador ante cambios dentro de los diferentes intervalos o *sprints*. En nuestro caso se realizarán reuniones periódicas que separaran los mencionados *sprints* en los cuales se mostrarán pequeñas entregas que incrementen la completitud del proyecto. Dichas entregas constarán de una planificación pequeña de las *historias de usuario* que se deberían tener completadas para la siguiente reunión, un diseño simple que se agregue al proyecto para soportar dichas historias, la codificación asociada al diseño y una serie de pruebas simples que sirvan para determinar si las historias funcionan.

Pese a que así se mencionen etapas que luego se verán en la planificación, se debe comentar que las tareas realizadas en las historias son las presentes en la planificación en el sentido de que dentro de las mismas podremos encontrar pequeños paquetes de trabajo que impliquen planificar, diseñar, implementar y probar. Por lo tanto no se deben de confundir las acciones realizadas en cada *sprint* con las tareas a alto nivel de diseño o pruebas presentes en la planificación.

Algunos de los motivos que han derivado en la elección de esta metodología son los siguientes:

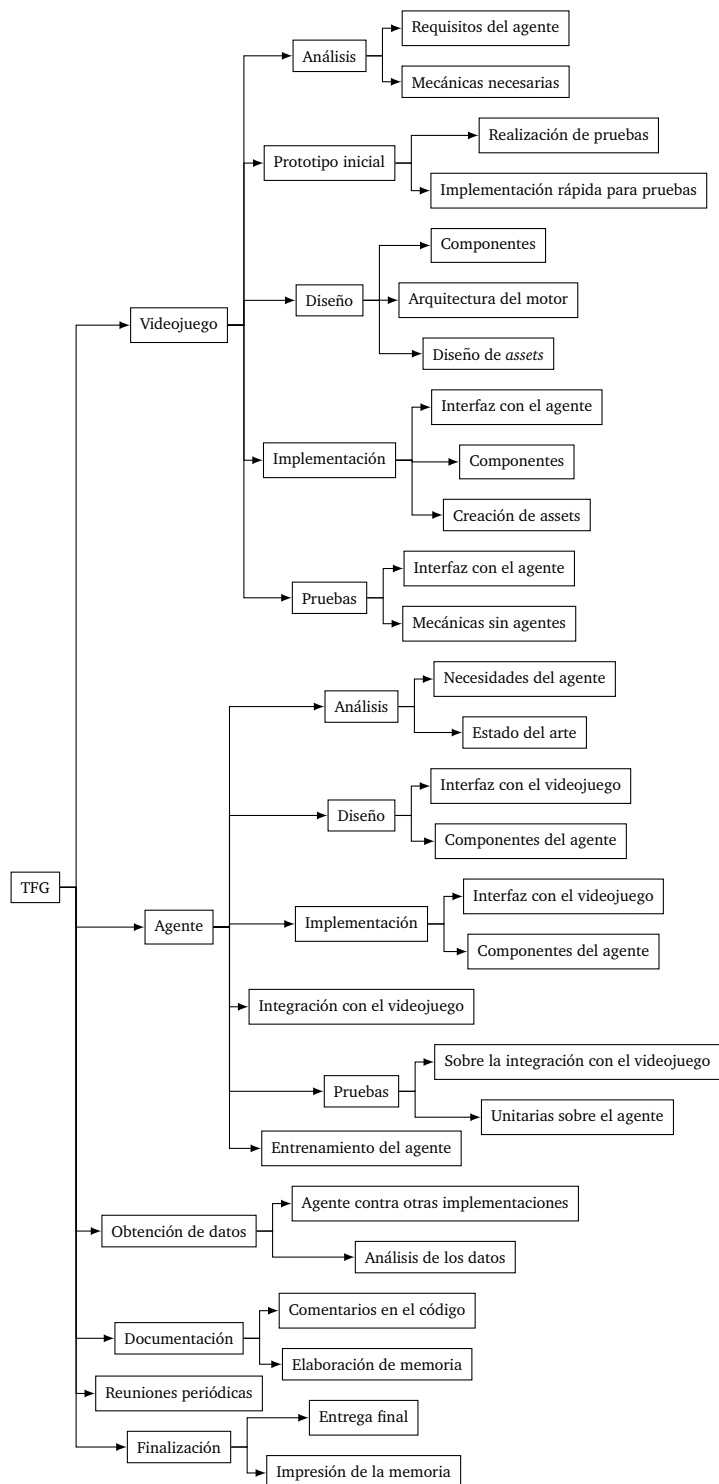
- **Rápida iteración:** que permite avanzar rápido en el proyecto manteniendo a los tutores informados.
- **Refactorización de código:** dado que se anima al programador a mejorar código anterior para evitar problemas futuros.
- **Bienestar del programador:** como el trabajador es el único en el equipo de desarrollo y es posible que tenga trabajo originado por otras fuentes es esencial que la metodología busque que el mismo se encuentre fresco y descansado.
- **Diseño simple:** ya que se considera que el mejor software es el que cumple con sus requisitos sin quedarse corto o sobrepasarlos realizando trabajo adicional innecesario.
- **Facilidad de división del proyecto:** haciendo que el trabajador se centre en partes claramente separadas y no tenga que abarcar el proyecto como un todo desde un principio.
- **Prototipos funcionales tempranos:** Al priorizar las historias de la aplicación se permitirá obtener un prototipo funcional temprano necesario dados algunos de los riesgos del proyecto.

4.4. Planificación temporal

Este apartado contendrá a grandes rasgos la planificación temporal del proyecto desarrollado. Para poner en contexto dicha planificación es necesario primero analizar el producto que se desea dados los objetivos y requisitos para identificar que tareas son necesarias llevar a cabo para completar obtener el producto deseado. Para este fin utilizaremos una herramienta comúnmente utilizada en gestión de proyectos como lo es la *Estructura de Descomposición del Trabajo* o EDT. Esta representación sirve para descomponer jerárquicamente el trabajo que requiere el proyecto.

En la Figura 4.1 se muestran las tareas que se han realizado. Para su obtención se ha seguido una estrategia de arriba hacia abajo o *top-down* en la cual se han identificado primero los paquetes de trabajo generales para luego ir desagregándolos en pequeñas tareas.

Figura 4.1: Estructura de Descomposición del Trabajo o EDT



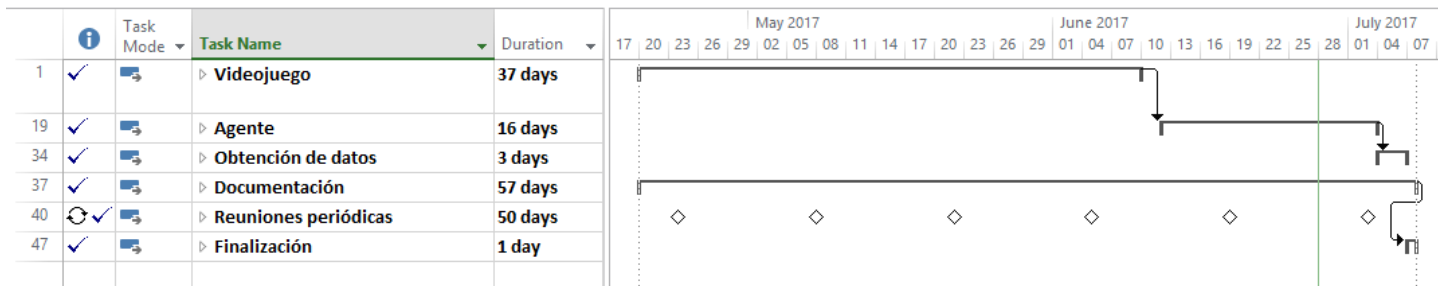


Figura 4.2: Cronograma general del proyecto

Para facilitar la comprensión de la planificación en su conjunto se muestra en la figura 4.2 el cronograma con las tareas de primer nivel solamente ya que mostrarlo completo no es posible en este formato de documento. Dedicaremos los siguientes subapartados del documento a explicar con detenimiento los grupos de tareas que en la figura 4.2 aparecen replegados mostrando su cronograma asociado.

4.4.1. Planificación del videojuego

En la figura 4.3 se puede observar un desglose de las tareas correspondientes a la creación del videojuego en todos los sentidos. Se comienza por una etapa de análisis en la que se intenta diseminar cuales son las **mecánicas** que se necesitarán implementar para dar una experiencia jugable lo suficientemente buena para luego centrarse en descubrir que **requisitos** puede que necesite la aplicación para acomodar al agente que luego se añadirá.

Seguido de esto se encuentra el apartado que forzó a rehacer parte de la planificación y que se relaciona estrechamente con el riesgo RSK-9 (Imposibilidad de realizar simulaciones aceleradas). Fue durante la ejecución de la tarea de **pruebas** en la cual se descubrió que la **implementación** con la tecnología anterior no podía cumplir los requisitos del proyecto.

La acción correctiva AC-1 para el riesgo RSK-9 hizo que fuera necesario agregar tareas de **diseño de componentes** y **diseño de arquitectura del motor** a la ya existente tarea de **diseño de assets**. Una vez terminada esta etapa de diseño se procede a la implementación de acuerdo con lo definido en la etapa anterior, esta etapa contiene la tarea más larga del proyecto al ser necesario **implementar los componentes** del motor desde cero, además de crear la **interfaz con el agente** y realizar la **creación de assets** que implicaría dibujar las animaciones, generar los sonidos, etc.

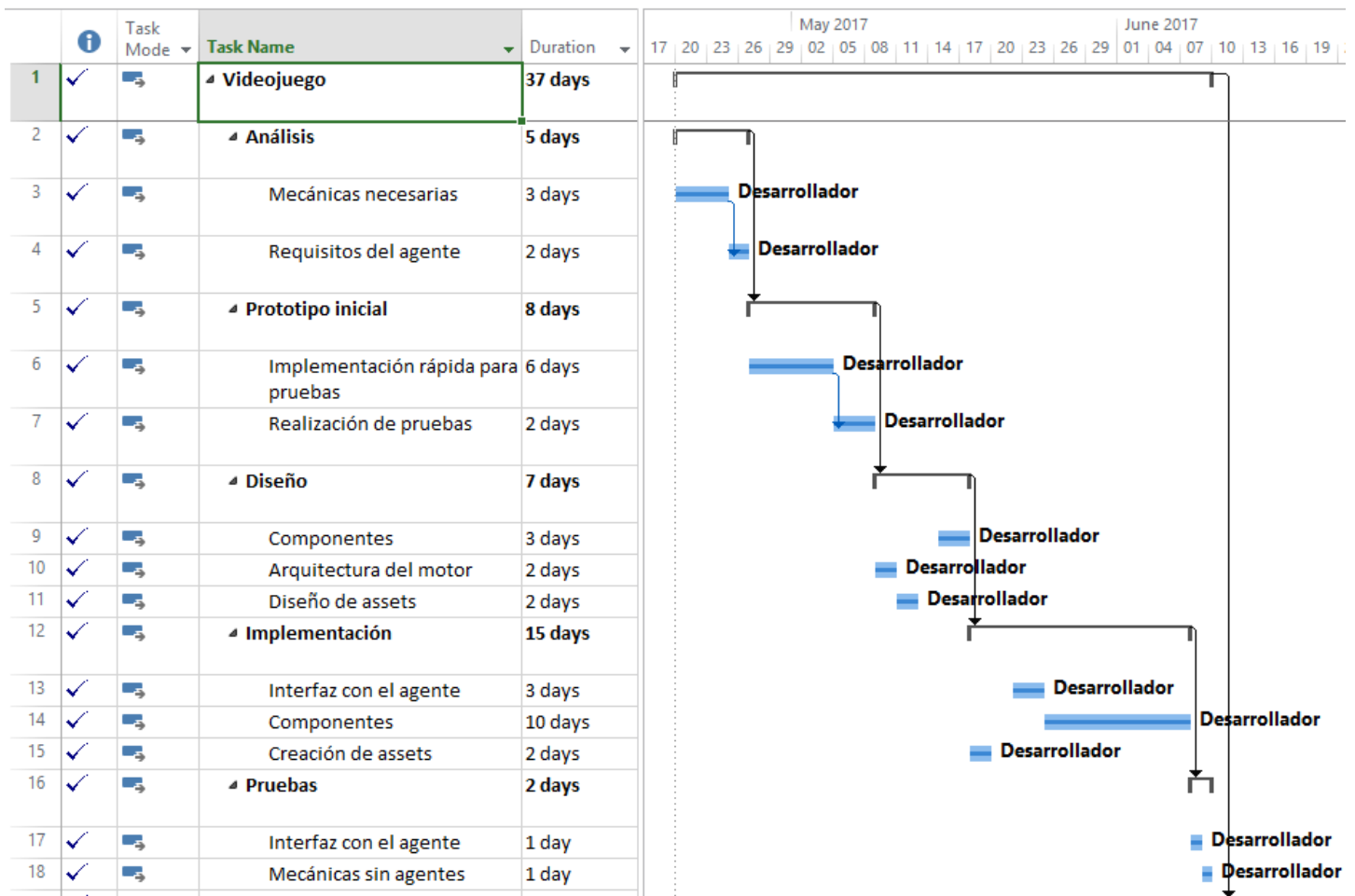


Figura 4.3: Cronograma de la parte del videojuego

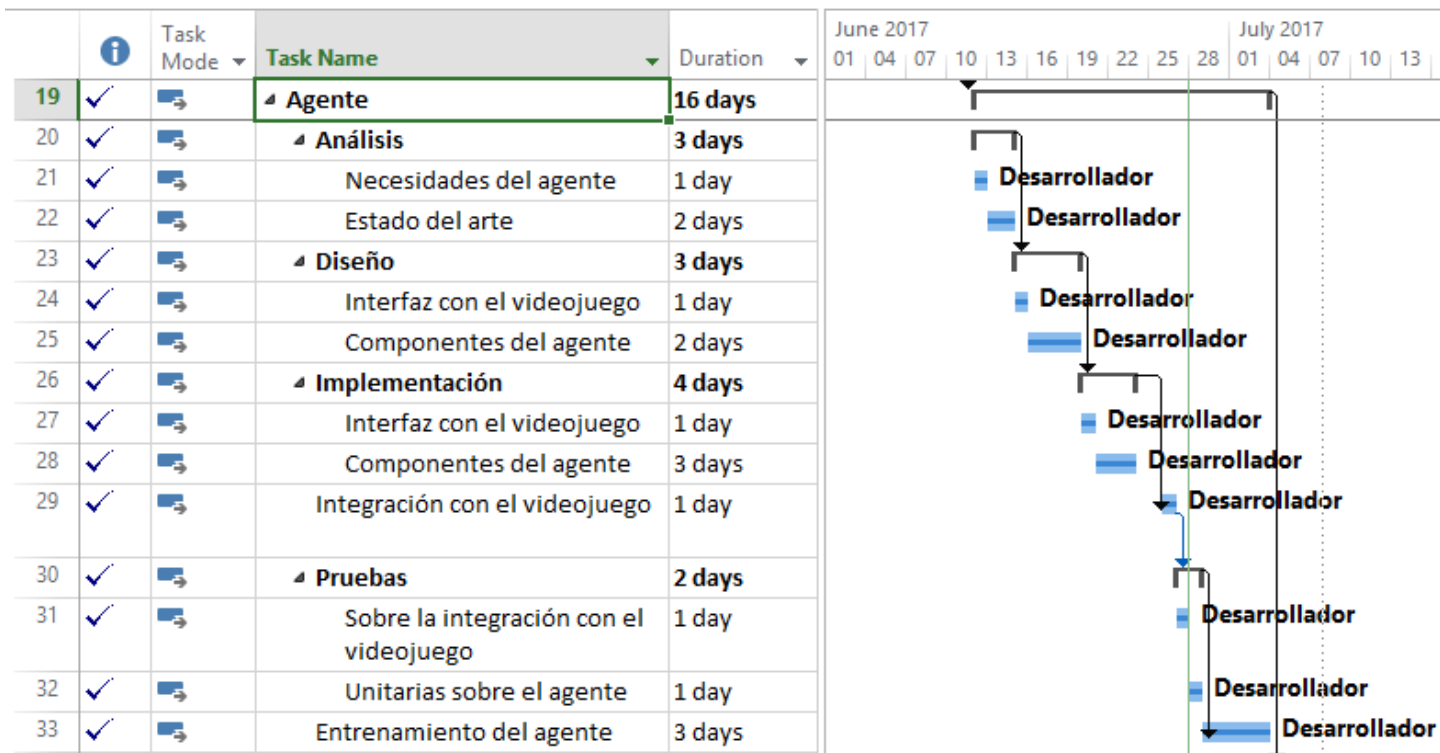


Figura 4.4: Cronograma de la parte del agente

Finalmente se realizan una serie de tareas dedicadas a pruebas en las que se comprueba el correcto funcionamiento de la posible interfaz con el futuro agente y las mecánicas generales del juego cuando aún no se ha añadido dicho agente.

4.4.2. Planificación del agente

Se dedica ahora este apartado a explicar la figura 4.4 que contiene la planificación asociada a todas las etapas que contienen las tareas que darán lugar al agente final. Es importante comentar que la duración de esta etapa había sido planificada de forma que era significativamente más larga. Sin embargo, la AC-1 hizo que se alargara más de lo inicialmente esperado la etapa dedicada al videojuego y redujo el tiempo disponible para el agente.

Se comienza, como era de esperar, por una etapa de **análisis** que nos ayudaría a obtener la información necesaria sobre lo que necesitaríamos para realizar una buena implementación, ya sea en forma de requisitos, riesgos que se deben considerar, etc. Primero se analizan las **necesidades de nuestro agente** dependiendo del juego para investigar el **estado del arte** para ver que

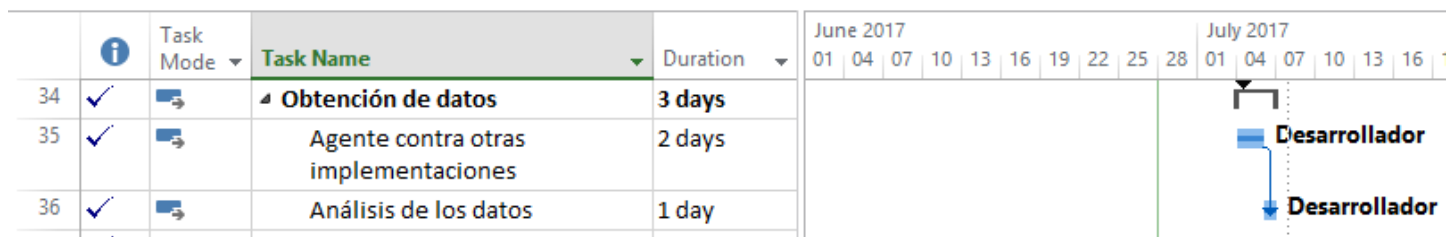


Figura 4.5: Cronograma de la parte de obtención de datos

información nos puede ser útil.

Dentro de las etapas de **diseño e implementación** se incorporan tareas dedicadas a definir e implementar tanto una **interfaz con el videojuego** creado como los **componentes** necesarios para que el agente funcione. Luego de estas dos etapas se realiza una **integración con el videojuego**.

Una vez que la aplicación está integrada se necesita hacer una serie de **pruebas** sobre dicha **integración** para ver que ambas interfaces cumplen su función. Además de eso, para comprobar el buen funcionamiento del agente se realizan una serie de pruebas **unitarias** sobre el mismo. Una vez superadas las pruebas se puede proceder a **entregar** al agente mediante la iteración de simulaciones que permite la aplicación.

4.4.3. Planificación de la obtención de datos

Esta pequeña sección contiene explica las tareas presentes en la figura 4.5. La primera tarea consiste en **probar al agente** una vez realizadas las tareas de aprendizaje. Para este fin se usa una implementación basada en reglas creada por el desarrollador y utilizando su conocimiento experto en el videojuego para comprobar como el agente es capaz de comportarse.

Se obtendrán datos sobre como se comporta con distintos niveles de entrenamiento y realizando diferente número de simulaciones, contra el mismo y contra el enemigo basado en reglas. Finalmente, los datos obtenidos se procesarán en la tarea de **análisis de datos**.

4.4.4. Planificación de la documentación, reuniones y finalización

Por último, en este apartado se pueden ver las tareas de primer nivel dedicadas a la **documentación**, las **reuniones** y la **finalización** del proyecto.

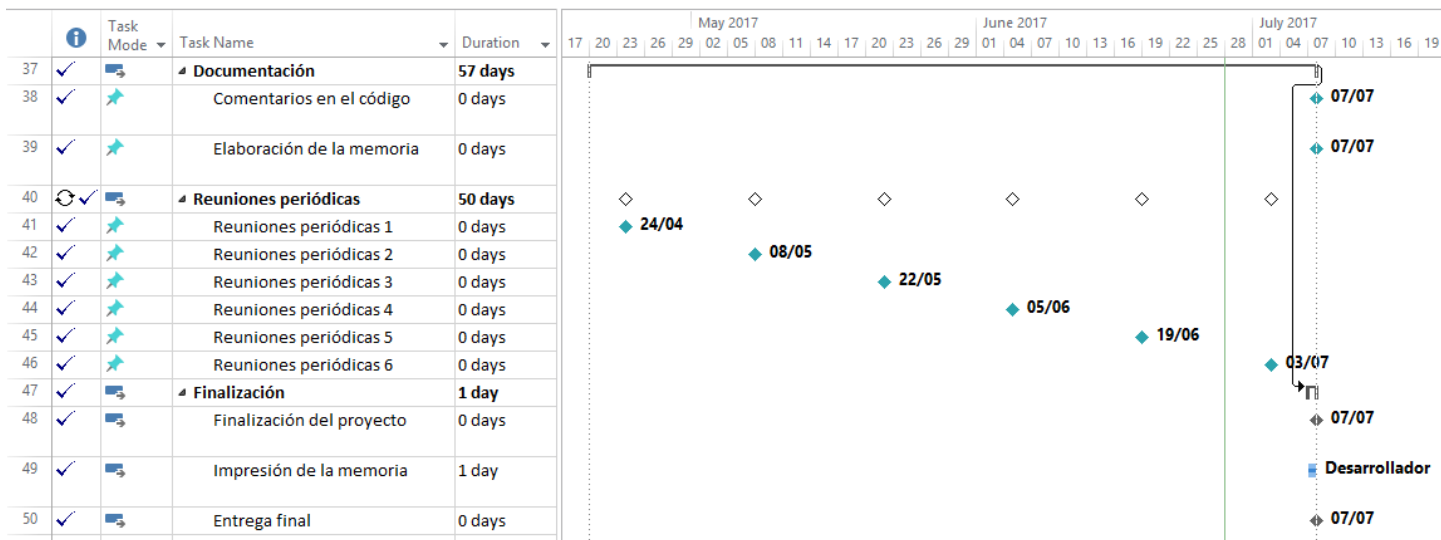


Figura 4.6: Cronograma de la parte de documentación, reuniones y finalización

Sobre la documentación hay que comentar que tanto la tarea de **comentarios en el código** como la de **elaboración de la memoria** se ha introducido una duración nula pues se han llevado a cabo con todas las otras tareas del proyecto. Esta es la razón por la cual se ha introducido la tarea de documentación en si misma como una tarea *hamaca* de todo el proyecto⁸. Si se introdujera una duración específica se estaría duplicando el trabajo en la planificación pues esas horas ya se han tenido en cuenta en las otras tareas del proyecto.

En lo que a las **reuniones periódicas** se refiere, se ha agregado una tarea recursiva que indica cuando se realizaron las reuniones con los tutores. Su duración, de un modo similar a las tareas de documentación, es nula porque el tiempo empleado se tiene en cuenta en las otras tareas. Si se le fuera a asociar una duración se tendría que hacer en términos de horas pues no se dedicaba todo un día a realizar una reunión. Utilizar horas en lugar de días suele ser un síntoma de haber desgranado las tareas demasiado y se considera mala práctica por cierta parte de la comunidad de gestión de proyectos.

Finalmente se agregan las tareas relacionadas con la **finalización** del proyecto, marcando un hito para dicha finalización, dedicando un día a **imprimir y comprobar la memoria** y dejando el último día como hito para realizar la **entrega final**.

⁸Una tarea hamaca es aquella cuya duración depende de cuando empiezan y terminan otras tareas del proyecto.

4.5. Análisis de costes

Este apartado estará dedicado a realizar un análisis de los costes asociados al proyecto con el objetivo de generar un presupuesto que nos permita controlar los gastos. Los tipos de costes que se considerarán son los siguientes:

- **Directos:** Los que se identifican claramente como costes definidos para el proyecto, un buen ejemplo es la compra de software o el pago por horas trabajadas.
- **Indirectos:** El PMBOK[1] ayuda a identificar los costes indirectos como aquellos que no se pueden asignar de forma directa a un proyecto concreto. Por esta razón se acumulan y distribuyen equitativamente entre diferentes proyectos mediante un procedimiento específico.

Pese a que el proyecto sea realizado en un ámbito académico por solamente un alumno no se debe restar importancia a este apartado ya que una buena gestión de costes es vital para que un proyecto salga adelante. Un buen ejemplo nos lo da el Chaos Report[7] que indica que recientemente el 50.7 % de los proyectos de software cuestan una media del 189 % más de lo esperado y que el 31.1 % de todos los proyectos se cancelarán antes de ser completados. Otro dato preocupante es el que indica que solo el 15.5 % de la totalidad de proyectos evaluados está por debajo de un 20 % de coste adicional.

Continuando ahora con el análisis de costes directos se ve que, al haber priorizado la utilización de software libre o versiones de prueba, los elementos a tener en cuenta son muy reducidos.

El más importante de los mismos es el salario del desarrollador para lo cual se ha investigado el valor medio pagado a un analista programador en Galicia. Dicha información se ha obtenido de un estudio salarial realizado por la consultoría Vitae[8] del cual se obtiene que el valor medio para un analista-programador junior de C++ es de 16000€ brutos anuales. A raíz de datos que se nos han proporcionado durante los estudios en la asignatura de Gestión de Proyectos obtenemos que los impuestos suman los siguientes porcentajes:

- 5.5 % de Seguridad Social
- 23.6 % de Contingencias Comunes
- 0.2 % de FOGASA
- 0.7 % de Formación Profesional

Bruto anual	Impuestos	Total anual	Total mensual	Coste/hora
16000€	4800€	20800€	1486€	9.30€

Cuadro 4.3: Tabla de salario del desarrollador

Meses de vida	Meses de utilización	Coste total	Coste en el proyecto
60	4	2923,36€	194.90€

Cuadro 4.4: Tabla de costes de amortización del equipo

Lo que corresponde a un incremento total del 30 % sobre el salario bruto que genera en la tabla 4.3 con los valores que nos llevan al coste/hora final.

Otro de los costes directos a considerar será el precio de impresión de la memoria y los materiales necesarios para crear el proyecto en formato CD. Teniendo en cuenta una longitud cercana a las 150 páginas y el precio de la grabación del CD se agregarán 100€ en concepto de ambos aspectos.

Considerando ahora los costes indirectos agregaremos el coste de amortización relacionado al equipo de desarrollo utilizado. El MacBook Pro utilizado costó en su día 2923,36€ y se le estimará una vida media de unos 5 años (60 meses). Dado que el proyecto se extendió aproximadamente 3 meses obtenemos los datos de la tabla 4.4.

Finalmente sumaremos un 10 % al sobre el coste del proyecto en concepto de gastos en servicios como electricidad, agua, internet, etc. Teniendo en cuenta el pago final⁹ al desarrollador con el resto de gastos directos e indirectos se obtienen los datos de la tabla 4.5 que nos muestra unos costes finales de 4672.14€.

⁹Considerando 14 pagas, 20 días laborales al mes con 8 horas de trabajo al día.

Elemento	Coste (€)
Analista programador C++	3952.50
Amortización del equipo	194.90
Xcode 9	0
Unity 5	0
Clang 4	0
Audacity 2.1.3	0
StarUML 2	0
Atom 1.19	0
Microsoft Project 2013	0
TeXstudio	0
Impresión memoria + CD	100
GraphicsGale 2.06	0
Dropbox	0
Google Drive	0
Git & GitHub	0
Costes indirectos	10 % (424.74)
TOTAL:	4672.14

Cuadro 4.5: Tabla de costes finales

4.6. Plan de Gestión de las Comunicaciones

Este apartado se dedicará a definir los protocolos dedicados al intercambio de información entre el equipo de desarrollo, en este caso el único trabajador con el rol de desarrollador, y los interesados en el proyecto que estarán compuestos por los tutores y posibles usuarios. Además se agregará la información correspondiente a cada uno de los interesados necesaria para comunicarse con ellos.

4.6.1. Gestión de interesados

El PMBOK[1] realiza la siguiente definición de interesados en el contexto de un proyecto:

Un interesado es un individuo, grupo u organización que puede afectar, verse afectado, o percibirse a sí mismo como afectado por una decisión, actividad o resultado de un proyecto. Los interesados pueden participar activamente en el proyecto o tener intereses a los que puede afectar positiva o negativamente la ejecución o la terminación del proyecto.

Para continuar con la gestión de las comunicaciones es esencial identificar primero a los interesados en el proyecto. Generalmente es un proceso complejo ya que su continuidad y solamente el número de posibles interesados en un proyecto de software pueden hacerlo realmente difícil. En el caso que nos atañe el número de interesados se reduce significativamente comparado con un proyecto software en un contexto de empresa. Esto se debe a que se está realizando un proyecto dentro de un entorno de formación e investigación, lo que elimina muchos de los posibles interesados que surgirían de entre los muchos posibles usuarios, clientes, proveedores, etc.

La tabla 4.6 contendrá la información relacionada con los interesados identificados que pueda ser necesaria realizar una comunicación.

4.6.2. Planificación de la información y comunicaciones

Esta sección se enfoca a lograr que los implicados en el proyecto dispongan en todo momento de la información que puedan necesitar. En busca de esto deberemos definir con precisión qué información puede ser necesaria, para quién, cuando se debe proporcionar, etc.

El PMBOK[1] define tres tipos de comunicación posibles:

- **Comunicación interactiva:** En la que las partes intercambian información de forma multidireccional. Es la manera más eficaz de asegurar comprensión común e incluye reuniones presenciales, llamadas, mensajería instantánea o videoconferencias.
- **Comunicación de tipo *push* (empujar):** En la que se envía la información a los receptores que puedan necesitarla sin tener que garantizar su recepción y comprensión. Incluye cartas, informes, correos electrónicos, comunicados, etc.
- **Comunicación de tipo *pull* (tirar):** En la que los receptores de la información son los encargados de acceder a la información que requieran según su criterio. Esto incluye sitios intranet, bases de datos de lecciones aprendidas o repositorios de conocimiento.

En el presente caso la comunicación ha sido principalmente interactiva dadas las reuniones presenciales y otro tipo de colaboraciones directas que hayan podido ocurrir. Aunque también han ocurrido instancias de comunicación *push* en la que el trabajador simplemente comunica cierta información a los tutores de forma unidireccional. Las características de la comunicación se encuentran en la tabla 4.7.

4.6.3. Reuniones

Dados los múltiples problemas que suele ocasionar la realización de reuniones con nula planificación se dedica este apartado a definir a groso modo las características que las mismas deberían tener.

Se necesitan cuidar detalles como clarificar el objetivo de la reunión o definir la duración de la misma. Además, el lugar en el que se llevan a cabo también es importante pues se deben evitar distracciones y facilitar la comunicación de los asistentes. En este sentido se han utilizado las salas del CiTIUS¹⁰ dado que los tutores ambos tienen un puesto en el centro y uno de ellos cuenta con un despacho que cumple las características necesarias.

Con respecto al tiempo, se extienden alrededor a una hora aproximadamente, siendo las mismas más cortas si se han tratado todos los temas necesarios y pudiendo ser más largas de forma extraordinaria si no se han podido tocar todos los puntos requeridos.

¹⁰Centro Singular de Investigación en Tecnologías de la Información de la Universidad de Santiago de Compostela (www.citius.usc.es/)

Identificación				Evaluación				Clasificación		
Nombre	Empresa/ Grupo	Localización	Rol en el proyecto	Información de contacto	Requisitos	Expectativas	Influencia potencial	Fase de ma- yor interés	Externo/ Interno	Apoyo/ Neutral/ Opositor
Tutores del proyecto	Tutores	Santiago de Compostela	Tutores	manuel.mucientes@usc.es pablo.rodriguez.mier@usc.es	Tutorización y resolución general de dudas	Cumplir todos los objetivos del proyecto	Alta	Todo el pro- yecto	Externo	Apoyo
Desarrollador del proyecto	Equipo del proyecto	Santiago de Compostela	Desarrollador	ruben@osor.io	Compromiso con el pro- yecto y calidad en la elabora- ción	Completar el proyecto con éxito en el tiempo estimado	Alta	Todo el pro- yecto	Interno	Apoyo
Jugadores potenciales	Usuarios	Indeterminada	Cliente	No aplica	No aplica	Observar el agente y disfrutar del videojuego	Baja	Finalización del proyecto	Externo	Neutral

Cuadro 4.6: Matriz de interesados del proyecto

Grupo	Propósito	Contenido	Nivel de detalle	Importancia	Periodicidad	Emisor	Receptor	Formato	Idioma	Métodos o tecnologías
Tutores	Informar sobre el estado del proyecto y presentar dudas asociadas	Información técnica	Alto	Alta	Establecida por las reuniones puntuales en la planificación y puntualmente de forma excepcional	Desarrollador	Uno de los tutores	Comunicación personal y electrónica	Español	Presencial o correo electrónico
Equipo de proyecto	Solicitar información sobre el proyecto y proporcionar recursos útiles	Solicitudes o aportación de recursos	Alto	Alta	Se realizan de forma puntual	Uno de los tutores	Desarrollador	Comunicación personal y electrónica	Español	Presencial, correo electrónico o mensajería
Jugadores potenciales	Conocer las preferencias y opiniones de los jugadores	Información sobre su opinión sobre el videojuego	Medio	Media	Se realizan de forma puntual al final del proyecto	Usuario/ Desarrollador	Desarrollador/ Usuario	Comunicación personal y electrónica	Español/ Inglés	Correo electrónico, mensajería y ocasionalmente presencial

Cuadro 4.7: Matriz de comunicaciones del proyecto

Capítulo 5

Arquitectura y herramientas

El IEEE define arquitectura como los conceptos fundamentales o propiedades de un sistema en su entorno que se encarnan en sus elementos, relaciones y los principios de su diseño y evolución [2]. Como referencia el propio IEEE, es el estándar ISO 12207 el que identifica un diseño arquitectónico y un diseño detallado y desgranado en lo que a un sistema se refiere. A partir de esto se obtiene que el diseño de la arquitectura de un sistema describe la estructura y la organización del mismo, es decir, se centra en los subsistemas o componentes que forman el sistema completo y las relaciones entre los mismos desde un punto de vista de alto nivel.

Contando ahora con una idea definida de cuales son los objetivos y requisitos de nuestro proyecto, nos centraremos en este capítulo en el diseño a alto nivel y en la arquitectura a la cual nos referimos en el párrafo anterior. Para ello utilizaremos representaciones gráficas a alto nivel del sistema y detallaremos los elementos arquitectónicos del producto y que tecnologías han sido usadas en los mismos así como las herramientas de las que hemos hecho uso para llevar a cabo el proyecto en su totalidad.

5.1. Arquitectura del sistema

La arquitectura del sistema completo estará estrechamente relacionada con el plan de contingencia realizado para el riesgo RSK-9 (AC-1). Dicho plan ha implicado crear una arquitectura dedicada a acoger el motor del videojuego ya que es necesario poder controlar todo el código de la aplicación.

Haciendo uso de conocimientos previos de alguno de los recursos citados en la bibliografía como el conocido *Game Engine Architecture*[9] se ha definido una estructura basada en subsistemas unidos mediante un bus de mensajes. Esta

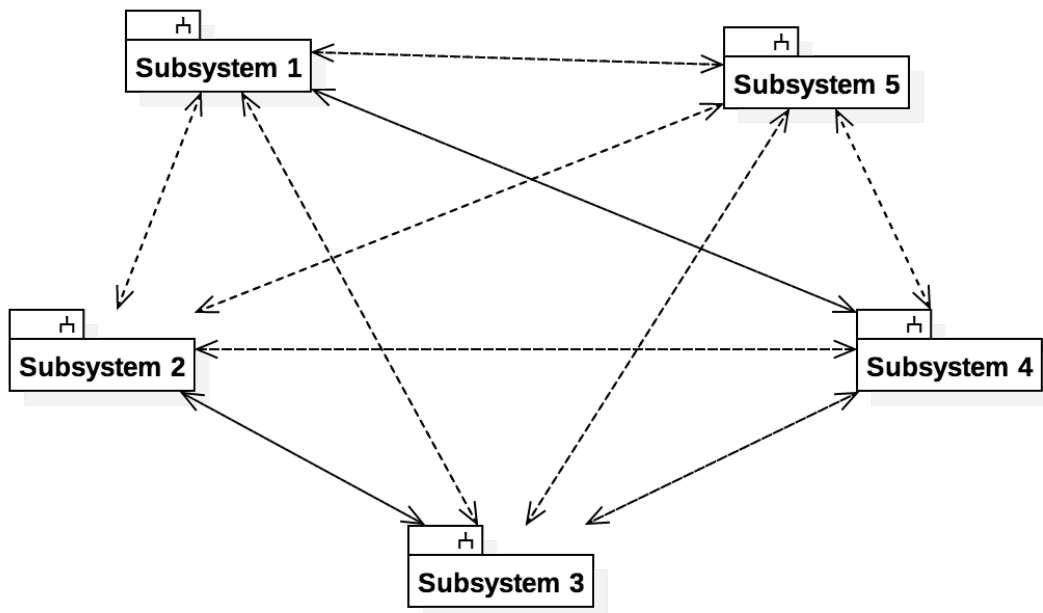


Figura 5.1: Arquitectura con subsistemas interconectados directamente

elección surge a raíz de la necesidad de separar claramente las implementaciones de todos los subsistemas y a la vez permitir que todos puedan comunicarse entre ellos. En aplicaciones de esta índole es muy común que todos los subsistemas tengan que poder referenciar a los otros. Se pueden dar multitud de situaciones en las que esto se requiera, un ejemplo podría ser que el subsistema de físicas detecte una colisión y se requiera reproducir un sonido cuando esto ocurra y mostrar una animación. Otros ejemplos podrían implicar al subsistema de inputs del jugador comunicándose con la lógica del juego, que a su vez actualizará lo que se ve por pantalla hablando con el subsistema de rendering y hará reproducir un sonido para indicar que la acción se ha realizado.

Es muy peligroso intentar mantener referencias a todos los otros subsistemas en cada uno de ellos ya que acabaríamos con una arquitectura en la que todo depende de todo y modificar una parte implica tener que lidiar con el código de prácticamente toda la aplicación. Los problemas de una arquitectura así quedan evidentes en la figura 5.1

Aquí es donde entra la arquitectura basada en subsistemas conectados por el bus de mensajes que permite evitar las múltiples referencias y controlar con

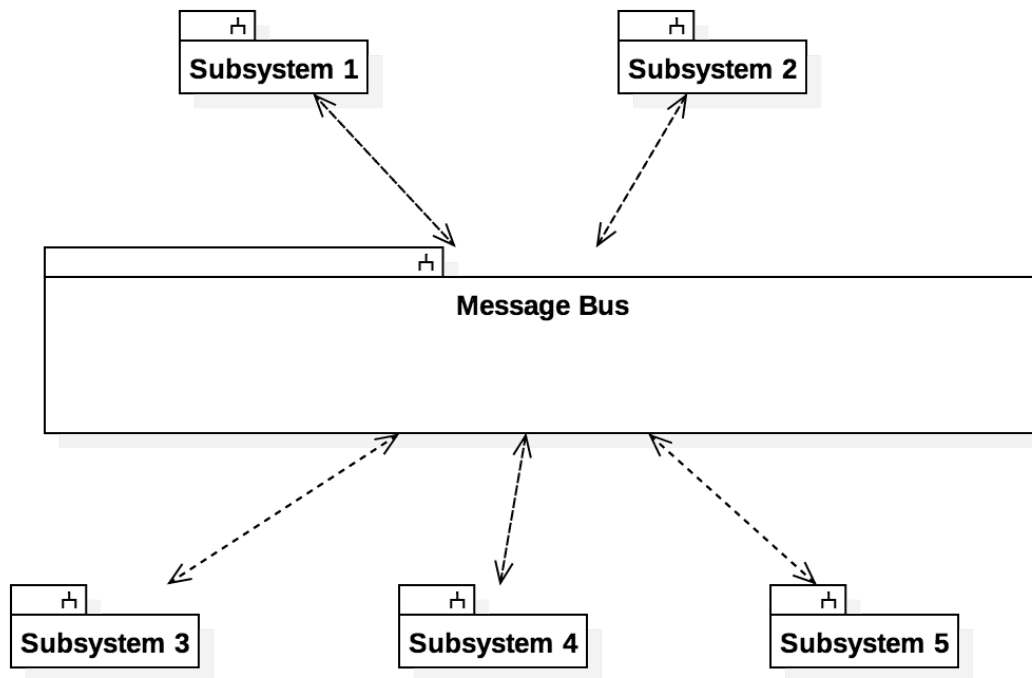


Figura 5.2: Arquitectura con subsistemas interconectados mediante bus de mensajes

mucha más facilidad las comunicaciones dentro de la aplicación. En la figura 5.2 se observa la diferencia en términos de simplicidad y orden.

De hecho, sobre la arquitectura general mostrada en la figura 5.2 se harán una serie de modificaciones de forma que todos los subsistemas sean totalmente transparentes a ojos del bus de mensajes. Para ello se hará que los mismos hereden de lo que llamaremos un *Bus Node* o nodo del bus. La forma que tendría esta arquitectura de forma genérica es la mostrara en la figura 5.3

Sobre esta arquitectura general se realizará una modificación relacionada con el requisito no funcional RNF-4 para facilitar los procesos de depuración de errores. Dado que todos los mensajes pasarán por el bus de mensajes se podrá hacer que los mismos sean mostrados en la consola interna de la aplicación o incluso introducir mensajes en forma de comandos en dicha consola. Para hacer esto posible se necesitará tratar de modo especial la consola como subsistema de forma que la arquitectura final será la que se puede ver en la figura 5.4.

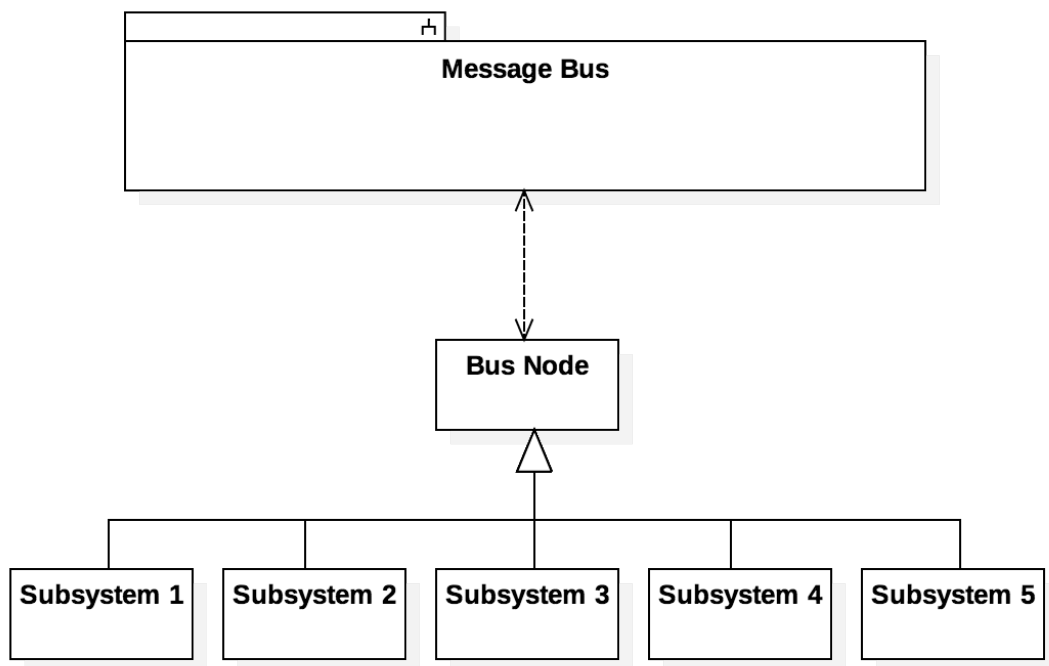


Figura 5.3: Arquitectura con Bus Node

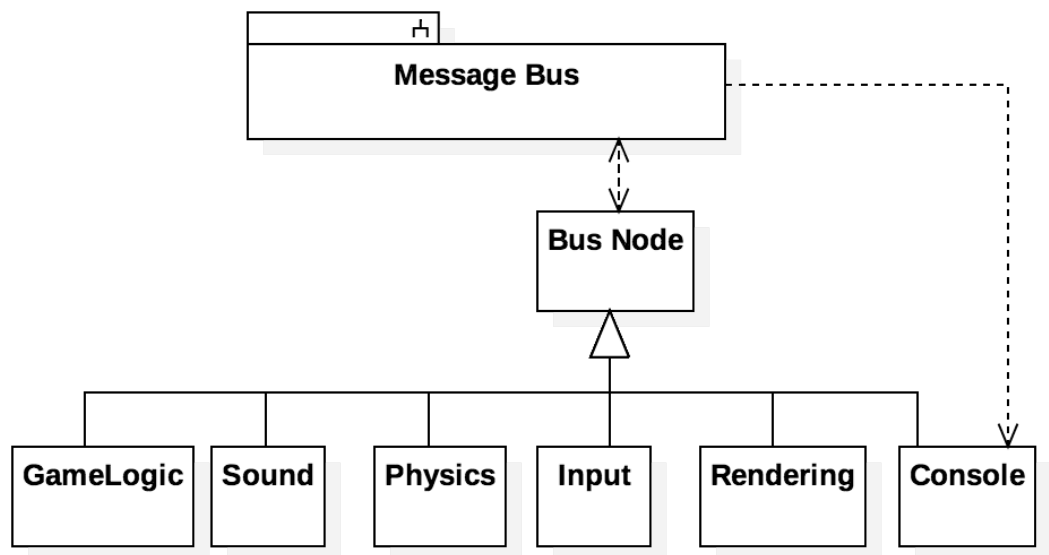


Figura 5.4: Arquitectura final

También podemos observar en la figura 5.4 el componente que representa la lógica del juego es la que ocasionará que se ejecuten las diferentes funcionalidades de una forma u otra. Como *GameLogic* se entiende a todas las clases dedicadas a definir los datos y comportamiento del videojuego en un determinado estado, por ejemplo, la lógica de un menú contendrá las opciones disponibles con la acción que corresponde a cada una de ellas, además tendrá la funcionalidad de cambiar entre las mismas y seleccionar la que se desee.

5.1.1. Subsistemas propios del motor

En relación a la figura 5.4 se dedicará este apartado a explicar la función general de cada uno de los subsistemas que no componen la lógica de la aplicación o *GameLogic*.

- **Sound o Sonido:** Será el subsistema encargado de reproducir cualquier tipo de sonido que necesite la aplicación, desde pequeñas indicaciones que confirman que se ha cambiado de opción en el menú hasta el sonido de realizar un ataque, incluyendo música si se fuera a añadir.
- **Physics o Físicas :** Será el subsistema encargado de lidiar con todas las funcionalidades relacionadas con la simulación de físicas en la aplicación. Esto puede incluir detectar colisiones contra muros y evitar que se puedan atravesar o detectar cuando un ataque realizado por un personaje ha alcanzado al otro o no.
- **Input o Entrada:** Será el subsistema que recibirá todas las entradas realizadas por el usuario, desde teclas pulsadas hasta movimientos de un joystick, para luego enviárselas a los otros subsistemas para que gestionen si deben ejecutar algún proceso.
- **Rendering o Renderizado:** Será el subsistema encargado de mostrar cualquier tipo de gráficos por pantalla. Necesitará gestionar la ventana y todas sus acciones (cambiar el tamaño, cerrarla, etc.) Además contendrá la funcionalidad de mostrar las animaciones realizadas, texto, efectos, etc.
- **Console o Consola:** Será el subsistema utilizado para ver los mensajes importantes de la aplicación así como enviar mensajes en forma de comandos introducidos en tiempo de ejecución. Será la interfaz que el usuario tendrá para introducir mensajes directamente en el bus de mensajes.

5.2. Herramientas de diseño

Los modelos formales mostrados en esta memoria utilizan UML o *Unified Modeling Language*. En términos de lenguajes de modelado para software este es sin duda el estándar más expandido, soportado además por el OMG¹/

Pese a que el lenguaje completo en su versión 2.5 ofrezca 14 tipos de diagramas específicos, en esta memoria se han utilizado solamente 4, siendo los mismos:

- **Diagrama de casos de uso:** Dedicado a modelar el comportamiento del sistema ante los actores que interactúan con él.
- **Diagrama de paquetes:** Utilizado para mostrar los subsistemas y componentes genéricos de la aplicación.
- **Diagrama de clases:** Usados para definir y especificar las diferentes clases de cada subsistema y las relaciones entre las mismas.
- **Diagrama de secuencia:** Que modelan los casos de uso definidos en términos de la interacción entre objetos de la aplicación.

5.2.1. Herramientas software

Relacionado con el diseño solo se ha utilizado una aplicación concreta siendo la misma el *StarUML 2.8*². La elección surgió a raíz de históricamente que ha sido la seleccionada para modelar durante la carrera y no se han experimentado deficiencias en términos de funcionalidades que implicaran tener que buscar un sustituto.

5.2.2. Patrones de diseño

Al trabajar con el paradigma de orientación a objetos es muy común que, si se diseña o programa sin prestar atención a las necesidades de la aplicación y sin considerar lecciones aprendidas anteriormente, se llegue a una situación en la cual se ha desarrollado una solución extremadamente compleja para un problema relativamente simple. A muy alto nivel, las lecciones aprendidas por parte de toda la comunidad de programadores y diseñadores se representan en forma de **patrones de diseño**.

¹*Object Management Group*: Consorcio dedicado a establecer y gestionar estándares relacionados con el paradigma de la orientación a objetos

²<http://staruml.io/>

Un patrón de diseño representa una solución a un problema común y genérico que ha sido generada y probada a partir de su uso continuado por parte de la comunidad. Lo que aporta esto es la flexibilidad de aplicar soluciones con la seguridad de estar utilizando modelos simples y potentes que no generarán errores en el diseño si se usan correctamente. Además, su implementación se hace significativamente más sencilla ya que la documentación y recursos disponibles para aprender a codificarlos está muy extendida.

En las siguientes secciones de este apartado se muestran los patrones utilizados de forma consciente en la aplicación ya que es relativamente común que debido a la práctica se lleguen a diseños que contienen patrones sin haberse propuesto implementarlos.

5.2.2.1. Patrón *Strategy*

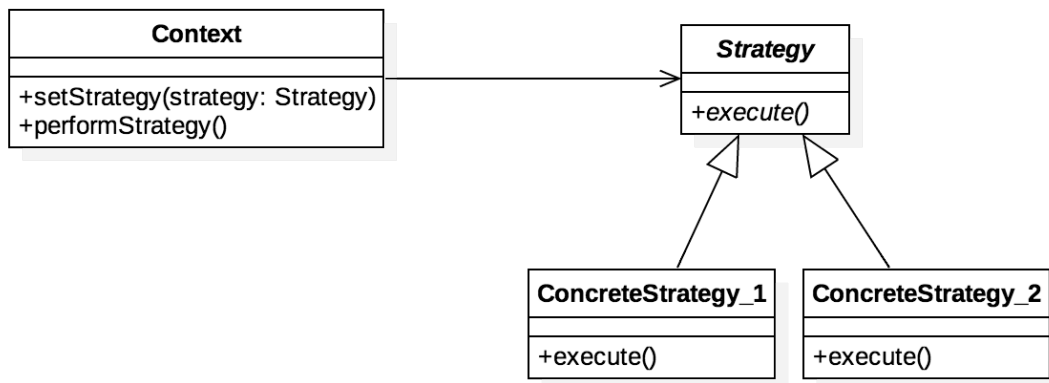
Para la implementación de comportamientos diferentes del enemigo nos será de mucha utilidad el patrón *Strategy* ya que el mismo está enfocado a cambiar los algoritmos que definen el comportamiento de un objeto de forma dinámica. De esta forma se puede cambiar fácilmente entre el enemigo controlado por el agente o el controlado por reglas siendo esto transparente al personaje que está siendo controlado.

Este patrón aporta una serie de ventajas relevantes para nuestro caso que son las siguientes:

- No se tiene que agregar el código de los algoritmos al cliente (personaje) ya que el mismo puede ser muy complejo.
- El cliente no necesita todos los algoritmos en todas las situaciones por lo que evitamos que sea él el que los almacene.
- Si existen varios clientes que utilicen los mismos algoritmos se evita duplicar el código.

En la figura 5.5 se observa una definición genérica del patrón donde las diferentes partes son:

- **Context:** La clase que contendrá a la estrategia sea cual sea la misma. Esto puede ser el propio cliente o puede estar contenida en él.
- **Strategy:** Clase abstracta que representa a una estrategia genérica, por si misma no contiene comportamiento alguno.

Figura 5.5: Patrón *Strategy*

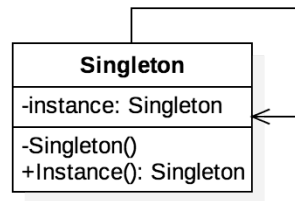
- **ConcreteStrategy:** Clase que implementa cada uno de los diferentes comportamientos entre los que se puede cambiar.

5.2.2.2. Patrón *Singleton*

A la hora de requerir que solo exista una instancia de un componente concreto en una ejecución del programa el patrón a utilizar es el conocido como *Singleton*. Este patrón permite además hacer que ese componente sea accesible por cualquier otro con facilidad, algo comúnmente necesario en videojuegos ya que hay entidades como gestores de recursos, relojes o sistemas de colisiones que deben ser comunes y únicos.

Estas dos ventajas comentadas son las que han decantado la balanza a favor de su elección. Sin embargo se debe tener mucho cuidado con el uso de este patrón ya que puede suponer una refactorización de código muy importante si luego se decide que el objeto no debe de tener acceso global o que se necesitan varias instancias del mismo.

En la figura 5.6 podemos ver el único componente que conforma el patrón. Esta clase tiene un método *Instance()* que permite acceder a la única instancia que existe. Dicha instancia o *instance* es privada, de la misma que el constructor *Singleton()* para evitar así que se creen nuevos objetos desde fuera.

Figura 5.6: Patrón *Singleton*

5.2.2.3. Patrón *Decorator*

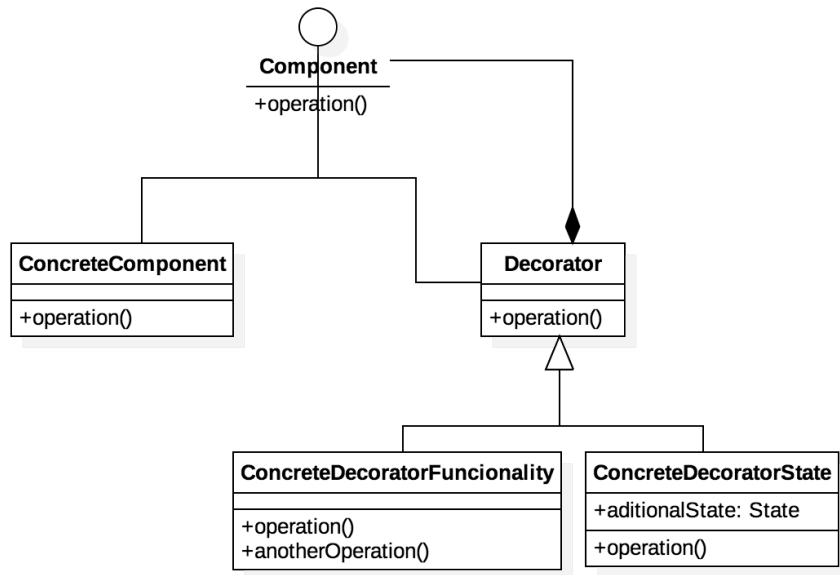
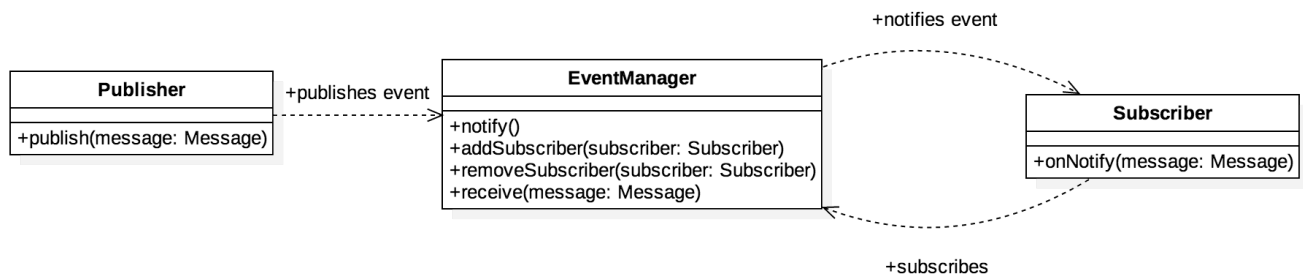
Cuando se necesita extender la funcionalidad de un objeto de forma dinámica el patrón más comúnmente usado es el conocido como *Decorator*. El mismo permite agregar componentes a un objeto de forma que los mismos puedan contener operaciones o estado adicionales. En el contexto de un videojuego suele ser necesario que la escena en la que se trabaje y los objetos que están en ella se puedan gestionar de forma dinámica. Se explicará más en detalle en el apartado de diseño e implementación.

En la figura 5.7 se ven los diferentes componentes que cuentan con las siguientes responsabilidades:

- **Component:** Interfaz para los objetos que pueden recibir nuevas responsabilidades dinámicamente.
- **ConcreteComponent:** Objeto que puede recibir responsabilidades adicionales de forma dinámica.
- **Decorator:** Referencia al objeto componente definiendo la interfaz que se ajusta a la interfaz del componente.
- **ConcreteDecorator:** Extienden el componente añadiendo funcionalidad y/o estado.

5.2.2.4. Patrón Publica-Subscribe o *PubSub*

El patrón Publica-Subscribe se utiliza cuando es necesario que diversos objetos reciban ciertos mensajes en los que pueden estar interesados mientras otros son los encargados de generar dichos mensajes. Esto se da en el caso del bus de mensajes ya que los diferentes subsistemas están interesados en recibir los mensajes relevantes para ellos.

Figura 5.7: Patrón *Decorator*Figura 5.8: Patrón Publica-Subscribe o *PubSub*

En la figura 5.8 se pueden observar los componentes del patrón que realizan las siguientes funciones:

- **Publisher**: Objeto que genera los mensajes y los envía al *EventManager*.
- **EventManager**: Objeto que recibe los mensajes y los distribuye a los suscriptores, además permite agregar o quitar suscriptores de forma dinámica.
- **Subscriber**: Objeto encargado de suscribirse que luego recibirá los mensajes por parte del *EventManager*.

Tenemos que comentar que en la aplicación, como se ve en el apartado de diseño, se realizará una aproximación ligeramente diferente en la cual todos los subsistemas publican y reciben mensajes. Estas dos responsabilidades estarán asociadas al *BusNode* que es el que se conectará con el bus de mensajes que representa al *EventManager*.

5.3. Herramientas de desarrollo

La elección de las herramientas de desarrollo utilizadas en este proyecto ha estado significativamente influenciada por el conocimiento previo que se tenía de las mismas. Esta práctica es habitual en todos los proyectos pero se vuelve más necesaria aún cuando el tiempo disponible para realizar el proyecto está limitado.

En este sentido, prácticamente siempre que se pudo elegir una herramienta conocida para realizar alguna tarea se ha escogido. Dicho esto, el entorno de desarrollo a rasgos muy generales que se ha utilizado está compuesto por las siguientes herramientas:

- **Sistema operativo**: macOS Sierra 10.12.5
- **Compilador**: clang 802.0.42 (basado en Apple LLVM 8.1.0)

El entorno es relativamente corto y fácil de describir dada la ausencia de bases de datos o frameworks que corran por encima del lenguaje utilizado. Pese a que en la lista anterior solo se mencionen el sistema y compilador utilizados, se entrará en detalle sobre las herramientas específicas dedicadas a la elaboración de la documentación, el desarrollo del código, las tecnologías elegidas y las librerías utilizadas.

5.3.1. Realización de la documentación

Para generar la documentación del proyecto y, por lo tanto, esta misma memoria se han utilizado las siguientes herramientas:

- **TeXstudio**³: La elección de este editor para archivos .tex se ha escogido dado su amplio uso en la comunidad, la cantidad y calidad de la documentación del mismo y el hecho de que es soportado en todos los sistemas operativos importantes, lo que reduce el riesgo de tener que utilizar otra herramienta si ocurren problemas con el equipo de desarrollo.
- **Microsoft Project 2013**: Gracias a las versiones de prueba que ofrece Microsoft para este software se ha podido utilizar esta herramienta. Teniendo la opción de elegir entre las diferentes opciones orientadas a la gestión de proyectos la mejor sin duda es Project. Aporta muchas más facilidades y posibilidades que los competidores y es el estándar de facto. Además, si las versiones de prueba dejan de estar disponibles, muchas de las otras herramientas soportan la importación de proyectos de Microsoft Project, posibilitando así continuar con el proyecto.
- **StarUML 2.8**: Usado tanto para elaborar los diagramas como para exportarlos a imágenes integrables en la documentación. Como

5.3.2. Desarrollo

Las herramientas que han sido necesarias para llevar a cabo la implementación de la aplicación han sido las siguientes:

- **Motor Unity 5**: Se escogió como herramienta principal para desarrollar la aplicación hasta que se descubrieron las limitaciones mencionadas en la acción correctiva AC-1. La elección inicial estaba fundamentada en las facilidades que otorga para implementar aplicaciones de este tipo y la consecuente velocidad a la hora de obtener versiones funcionales rápidamente.
- **IDE MonoDevelop**: Necesario para trabajar con el código en C# que utiliza Unity en macOS. Pese a que sea un IDE simplemente se utilizaba para la edición de código dadas las facilidades a la hora de detectar errores en el mismo.
- **IDE Xcode 8.3.3**: Al estar estrechamente integrado con el sistema de Apple nos permite realizar todo el desarrollo en un mismo entorno. Automáticamente busca y utiliza las herramientas por defecto a la hora de detectar

³<http://www.texstudio.org/>

errores en el código, compilar, depurar y empaquetar la aplicación. Además integra un potente *profiler* que ayuda a detectar los cuellos de botella de la aplicación, algo muy importante en situaciones como esta en la que el rendimiento es importante.

- **Compilador clang-802.0.42 basado en Apple LLVM 8.1.0:** La elección de este compilador era clara, además de soportar las especificaciones de C++ necesarias (completamente hasta C++14) también puede compilar código escrito en Objective C, ya que se ha requerido para una pequeña parte relacionada con el empaquetado de la aplicación. Este compilador es automáticamente utilizado por Xcode por lo que a ojos del usuario es completamente transparente.

5.3.3. Tecnologías

Las tecnologías que se incluyen en este apartado están compuestas por los diferentes lenguajes de programación escogidos para cada una de las partes, así como de cualquier otro tipo de lenguaje que se haya tenido que utilizar.

- **C#:** Elegido porque es el lenguaje que utiliza Unity para correr los diferentes archivos o *scripts* que definen el comportamiento de los objetos del juego.
- **C++:** Dada la relativa experiencia que el desarrollador tenía en este lenguaje se consideró el mismo una buena opción para el proyecto. Más incluso cuando se consideró que el rendimiento era importante dado que la ausencia de un *runtime* encargado de correr nuestro código, realizar recolección de basuras, etc podía ser catastrófico a la hora de conseguir la velocidad de simulación deseada. Además, en términos de lenguajes compilados a bajo nivel C++ es el que más se ajustaba al paradigma orientado a objetos.
- **Objective-C:** Extensión de C orientada a objetos similar a C++ pero utilizado principalmente en sistemas Apple. Nos ha permitido encapsular todos los archivos y recursos de la aplicación en un mismo ejecutable, haciendo el producto final extremadamente portable en todos los sistemas macOS.
- **JSON:** Formato de texto orientado a la transmisión y representación de datos de forma ligera. Se ha utilizado para escribir los datos referentes a las animaciones para así saber que imagen se debe mostrar, en que orden y durante cuanto tiempo.

5.3.4. Librerías

Finalmente, las librerías utilizadas simplemente se referirán a las que son llamadas desde el código C++ pues los otros lenguajes no requerían extensión alguna para realizar las funcionalidades que se implementaron con ellos.

- **Biblioteca estándar de C++:** Básica a la hora de trabajar con C++ ya que soporta los contenedores genéricos y funcionalidades asociadas más utilizados en el lenguaje a la vez que completa las características del mismo al soportar funcionalidades en entrada y salida a archivos o salida estándar.
- **SFML:** Librería a bajo nivel que sirve como interfaz sobre OpenGL. Gestiona la creación de ventanas en el sistema operativo que se requiera y ofrece funcionalidades útiles a la hora de mostrar por pantalla, reproducir sonidos, etc.
- **json.h:** Librería en formato de cabecera única que sirve para facilitar la lectura y escritura de archivos json. Disponible en GitHub⁴ en la cuenta de su creador, *sheredom*, el cual la proporciona sin requerir ningún tipo de licencia.
- **Boost:** Conjunto de bibliotecas de software libre enfocadas a extender al lenguaje C++ con funcionalidades comúnmente necesitadas como realización de operaciones algebraicas complejas, procesamiento de imágenes, expresiones regulares o nuevos tipos de datos.

⁴<https://github.com/sheredom/json.h>

Capítulo 6

Diseño e implementación

En este capítulo se documenta el diseño e implementación que se ha realizado para dar lugar a la aplicación final. Como bien se explica en el capítulo dedicado a la arquitectura, se ha seguido una división por subsistemas por lo que la mejor aproximación para separar ahora los múltiples diagramas de clases es hacerlo por paquetes, asociados cada uno de ellos a un subsistema. Además se generarán una serie de paquetes que contienen una serie de funcionalidades adicionales utilizadas por los otros.

Se empezará por la documentación asociada a los paquetes relacionados con el motor del videojuego en sí mismo para luego seguir con el subsistema de la lógica de la aplicación que contendrá la implementación del agente. Luego se muestran los diagramas de interacción que relacionan las clases antes definidas con los casos de uso en los que participan. Finalmente se mostrará como se llegó al diseño de la interfaz gráfica de la aplicación. Además, se ha hecho uso de un **código de colores** que permite una comprensión más rápida de cada uno de los diagramas:

- **Verde:** Identifica las clases protagonistas o principales de cada diagrama.
- **Morado:** Identifica las clases que ayudan a las principales a realizar la función que se les requiere, estas clases **pertenecen** al paquete/diagrama que se está mostrando.
- **Amarillo:** Identifica las clases que son necesarias para llevar a cabo las funcionalidades del paquete/subsistema pero que **no pertenecen** al mismo.
- **Azul:** Identifica las clases, estructuras o enumeraciones que representan contenedores de datos sin funcionalidades complejas.

6.1. Diagramas de clases

Para facilitar la comprensión de la estructura de esta sección la misma está ordenada de la siguiente forma:

1. Primero se explicará la estructura asociada al bus de mensajes.
2. Luego se mostrará la estructura genérica de la aplicación con todos los subsistemas.
3. Se explicarán los subsistemas individualmente.
4. Brevemente, se mencionarán algunas clases adicionales.
5. Finalmente se mostrarán los diagramas de cada una de las escenas.

En los apartados que siguen se añadirá una breve explicación asociada a cada diagrama para favorecer su comprensión. Además, en aras de mantener la simplicidad de los diagramas, se han omitido los constructores que no aportan información sobre la clase, es decir, si una clase no requiere argumentos o requiere los mismos que una superclase no se mostrará su constructor. En situaciones donde sea relevante como que requiera argumentos inesperados o el constructor sea privado se especificará como tal.

6.1.1. Bus de mensajes

En la figura 6.1 se encuentra el diagrama de clases del bus de mensajes, en el mismo se vé implementada una modificación del patrón *PubSub* (mostrado en la figura 5.8) en la cual la clase **BusNode** representa tanto a los publicadores como a los subscriptores. Dicha clase es la encargada de encapsular las funcionalidades asociadas a enviar y recibir mensajes por parte de cada subsistema.

Por otra parte, la clase **MessageBus** es la que representa el *EventManager* del patrón *PubSub*. La misma es la encargada de almacenar los mensajes y enviarlos a todas las instancias de *BusNode* que lo requieran.

Finalmente, se ven las clases **Message** y **MessageData** que representan los mensajes que se envía y su contenido.

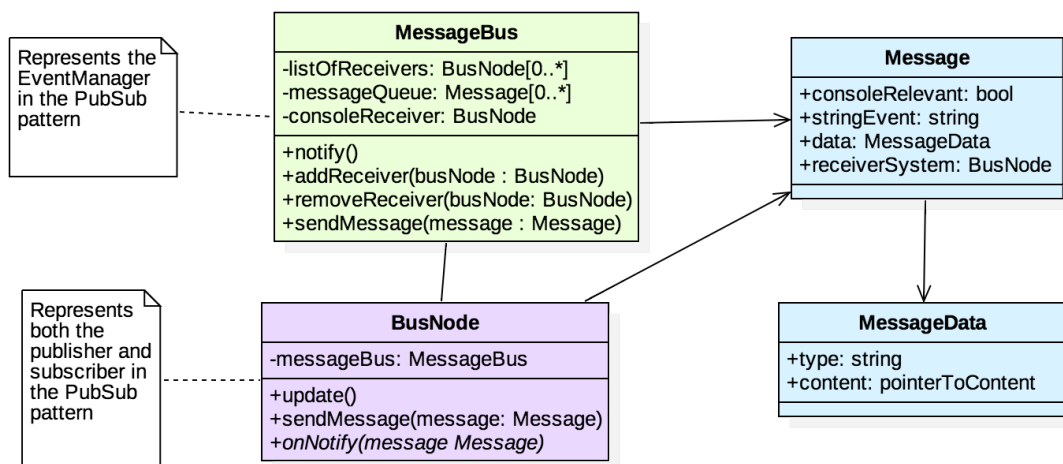


Figura 6.1: Diagrama de clases del bus de mensajes

6.1.2. Aplicación general

La figura 6.2 contiene la lógica del juego mostrada a muy alto nivel. La clase principal es **Game** que es instanciada en el *main* de la aplicación. La misma contiene la función *loop* que simplemente itera por todos los subsistemas, actualizándolos, como se ve en el diagrama de secuencia 6.13.

También se observan las clases asociadas a cada uno de los subsistemas, muy importante es la clase **GameState** que contiene a la escena de la aplicación representada por la clase **Scene**. Aquí podemos ver el patrón *Decorator* (Figura 5.7) en el cual la escena y los *gameobjects* (representados en la clase **GameObject**) agregan estado y funcionalidad. Cada *gameobject* representa uno de los objetos pertenecientes a la escena desde el punto de vista del juego a alto nivel, esto puede ser un personaje, texto, una barra de vida, un contador de tiempo, etc.

6.1.3. Subsistemas

Subsistema de consola

En la figura 6.3 se ven las clases que componen el subsistema de la consola. En el mismo, la clase **Consola** es la que contiene todas las funcionalidades necesarias, como abrirse y cerrarse con el método *toggleOpen*, ser dibujada con el método *draw* o evaluar la cadena de texto introducida con *evaluateInputLine*.

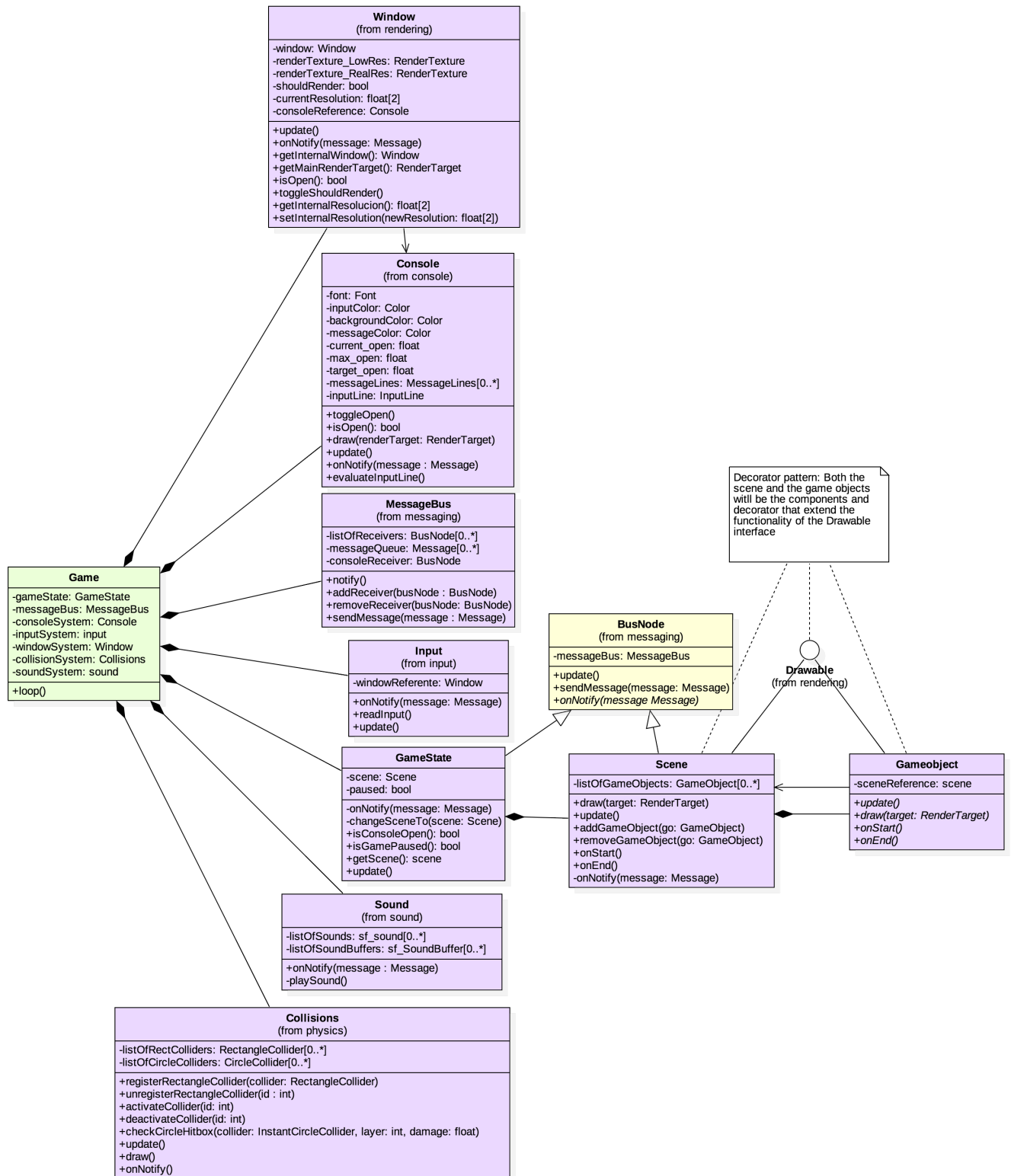


Figura 6.2: Diagrama de clases de la lógica del juego

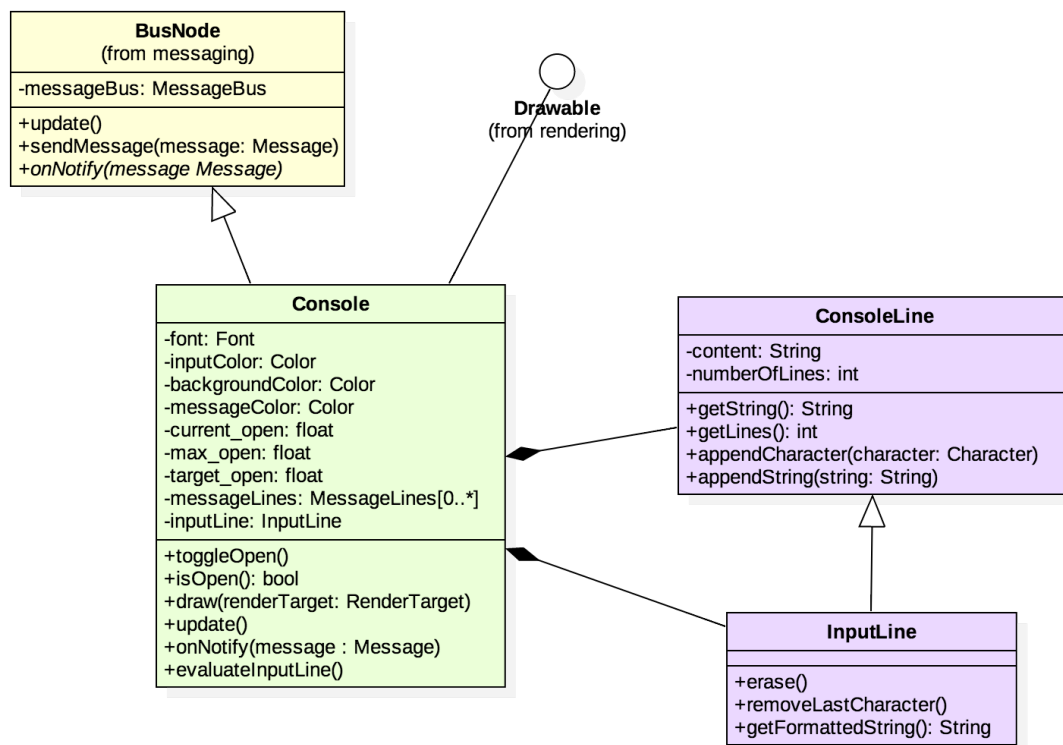


Figura 6.3: Diagrama de clases del subsistema de consola

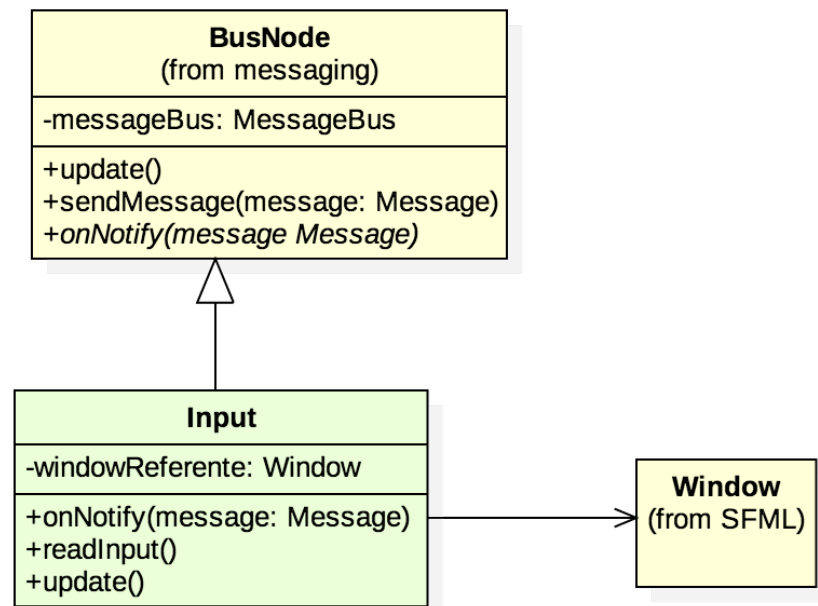


Figura 6.4: Diagrama de clases del subsistema de entrada

Vemos además la jerarquía de *ConsoleLine* e *InputLine*. Se encargan respectivamente de contener todas las líneas de texto a mostrar en la consola y de contener el texto introducido por el usuario. *InputLine* extiende a *ConsoleLine* en el sentido de que agrega las funcionalidades necesarias para agregar texto, eliminarlo y ser mostrara con un formato diferente.

Subsistema de entrada

La figura 6.4 muestra las clases referentes al subsistema de entrada. En dicho diagrama se puede observar como su definición es muy sencilla ya que solamente tendrá que iterar sobre todos los eventos de entrada que ocurren y enviarlos al bus de mensajes.

La clase principal *Input* tendrá una asociación con la clase *Window* de *SFML* ya que es dicha clase la que genera todos los eventos. Internamente, dentro de la función *update*, se irán recogiendo los eventos y enviándolos al bus de mensajes con el mensaje adecuado.

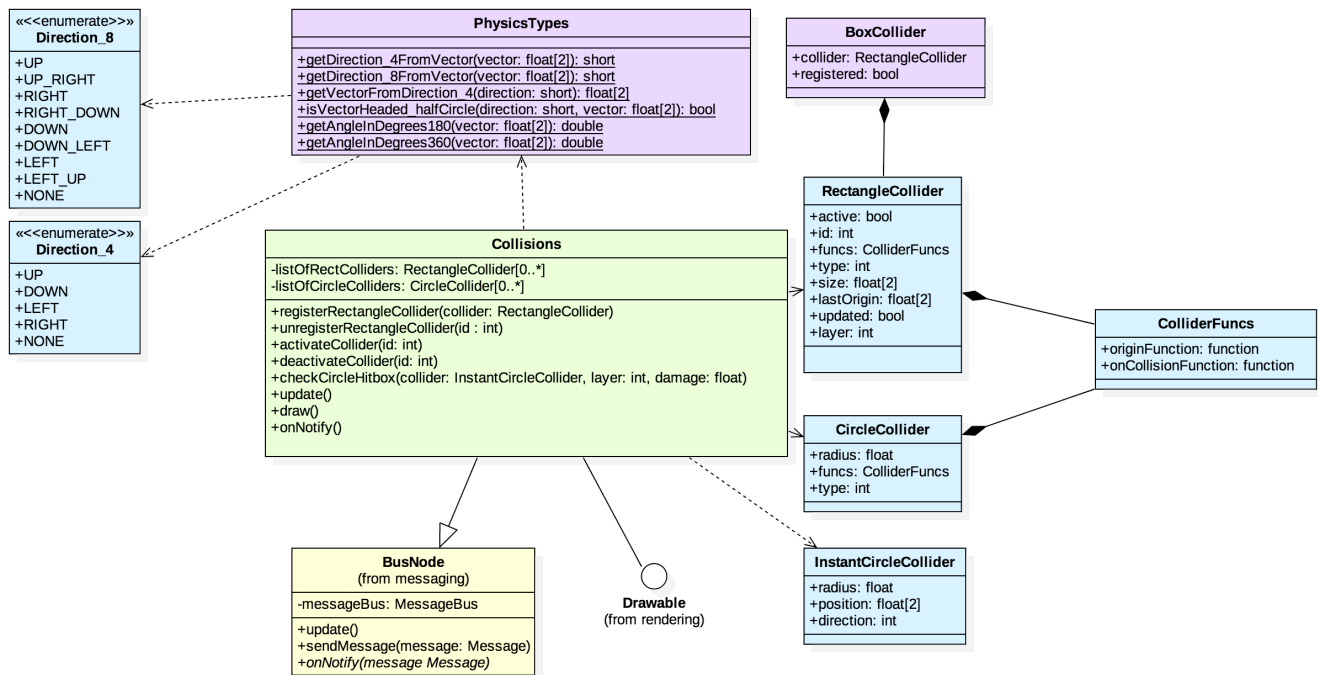


Figura 6.5: Diagrama de clases del subsistema de físicas

Subsistema de físicas

El subsistema de físicas está representado en el diagrama de clases de la figura 6.5. En el mismo destaca la clase **Collisions** que se encarga de guardar los **RectangleColliders** y **CircleColliders** de la aplicación. Estos *colliders* representan las diferentes figuras que pueden sufrir colisiones dentro de una escena y pueden tener forma de rectángulo o círculo. Ambos *colliders* también contienen referencias a funciones capaces de proveer la posición actual del collider y que contienen la función a ejecutar si ocurre una colisión. Dichas funciones las contiene la estructura **ColliderFuncs**.

InstantCircleCollider es una estructura que permite comprobar instantáneamente si un círculo arbitrario colisiona con algún otro *collider*. La inmediatez que esto permite es especialmente importante para comprobar si un ataque de un personaje ha acertado o no. Por otra parte, **BoxCollider** es una clase que encapsula un **RectangleCollider** para hacer transparente su creación y registro a otras clases de la aplicación.

Finalmente, **PhysicsTypes** contendrá estructuras que representan direcciones en 4 u 8 sentidos (**Direction_4** y **Direction_8** respectivamente) y apostará funciones útiles para trabajar con ellos.

Subsistema de renderizado

El subsistema de renderizado, mostrado en la figura 6.6, contiene la clase principal **Window** encargada de mostrar por pantalla la aplicación. Para ello aporta funcionalidades como:

- Parar de mostrar o volver a mostrar la aplicación con *toggleShouldRender*.
- Cambiar la resolución interna de la aplicación independientemente de la externa con *setInternalResolution*.
- Comprobar si la ventana sigue abierta para terminar la aplicación.

Para dibujar por pantalla hará uso de una serie de atributos, muchos de ellos aportados por la librería *SFML*:

- **window**: Objeto de la clase *Window* de *SFML* que representa la funcionalidad más básica de una ventana en el sistema operativo en el que se está ejecutando.

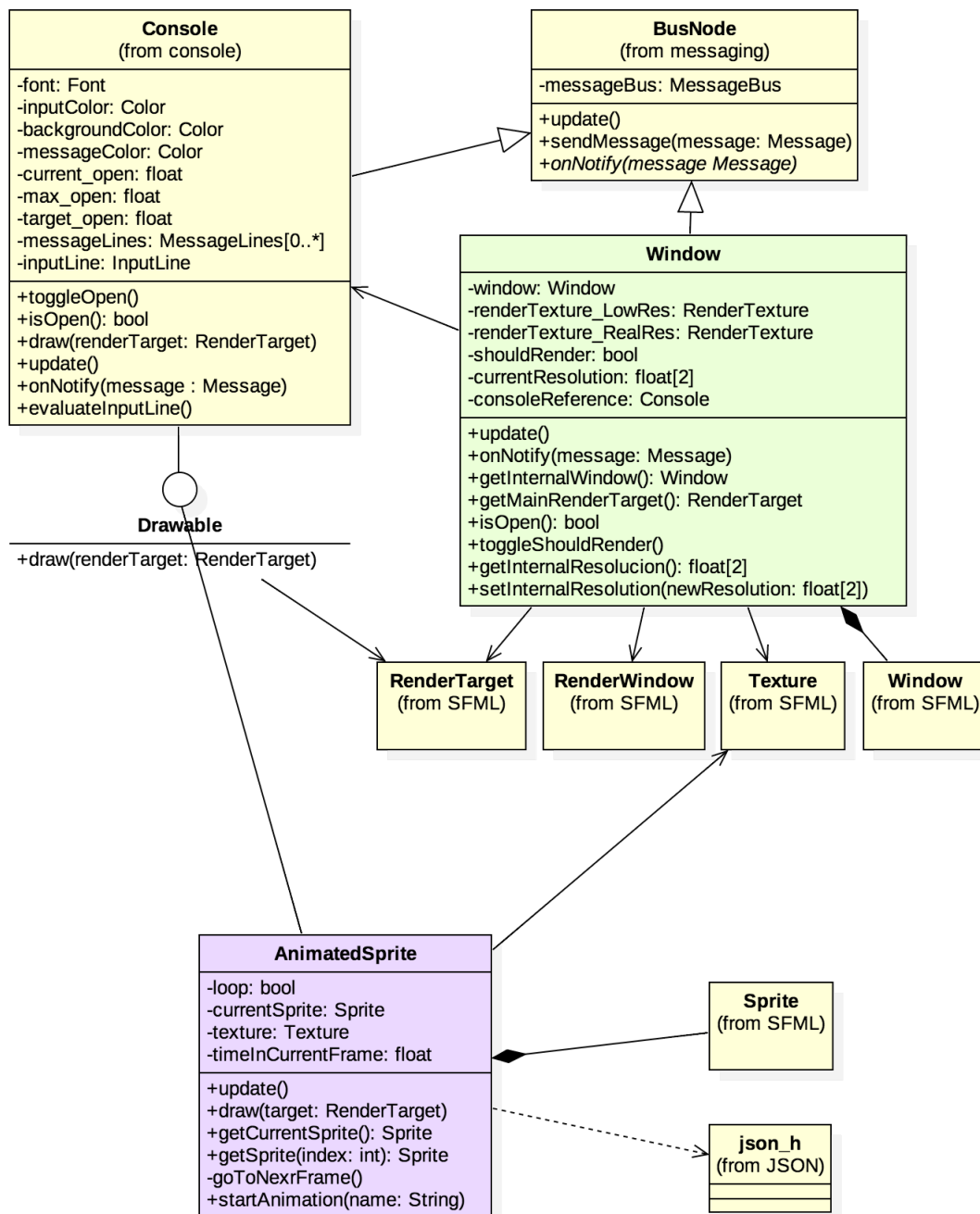


Figura 6.6: Diagrama de clases del subsistema de renderizado

- **renderTextures**: Dos objetos de la clase *RenderTexture* de *SFML* que representan texturas sobre las que se dibuja, una de ellas estará a baja resolución y se pintará el juego sobre ella para ahorrar recursos. Luego se escalará esa misma textura y se pintará por encima de la de alta resolución que es la que finalmente se muestra por pantalla.
- **shouldRender**: Booleano que define si se muestra o no algo por pantalla.
- **currentResolution**: Que guarda la resolución real de la ventana.
- **consoleReferente**: Referencia a la consola que permite pintarla siempre que sea necesario, independientemente de que exista una escena o de que el juego se encuentre detenido.

Además, la clase *AnimatedSprite* guarda una textura en formato png que contiene todos los dibujos de un personaje en las diferentes posturas que forma una animación, lo que se conoce como *spritesheet*. Para mostrar el dibujo correcto se lee previamente un archivo en formato JSON con ayuda de *json.h* que contiene las animaciones y donde están localizadas cada imagen individual dentro de la textura general.

Esta clase ofrece una forma general de guardar y mostrar movimiento de cara a *gameobjects* con animaciones.

Subsistema de renderizado

Finalmente, se puede observar el diagrama del subsistema de sonido en la figura 6.7. El mismo es muy sencillo, ya que los sonidos (representados por la clase *Sound*) unicamente son cargados al inicializar el subsistema, junto con los *buffers* (representados por la clase *SoundBuffer*) necesarios para su reproducción que guardan datos referentes a cada *tick* dentro del archivo de sonido.

Internamente, se utilizarán dos listas con dichos sonidos y *buffers* que serán accedidos cuando se utilice la función *playSound* para comenzar la reproducción de un sonido. La única forma de que se ejecute *playSound* es que llegue un mensaje concreto para cada sonido desde el bus de mensajes.

6.1.4. Otros útiles

Paquete de recursos

La clase *ResourceManager* mostrada en la figura 6.8 es la encargada de hacer que todos los recursos que necesita el juego estén disponibles en tiempo de

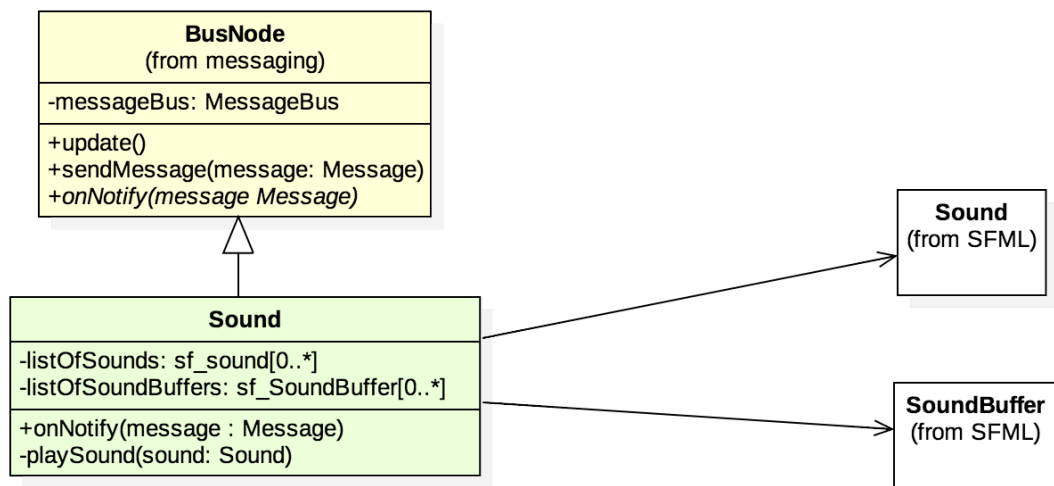


Figura 6.7: Diagrama de clases del subsistema de sonido

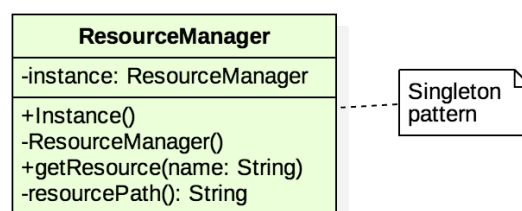


Figura 6.8: Diagrama de clases del paquete de recursos

Clock
-instance: Clock -deltaTimeList: float[0..*] -timeScale: float
+Instance() -Clock() + setFrameSeparator() + getDeltaTime(): float + getMeanDeltaTime(): float + getMeanFPS(): int + setTimeScale(scale: float) + getTimeScale(): float

Figura 6.9: Diagrama de clases del paquete de útiles

ejecución. Para lograr esto se ha implementado un *Singleton* que contiene partes en el lenguaje *Objective-C*. Esto permite trabajar directamente con el sistema operativo de Apple y acceder a recursos embebidos dentro del ejecutable *.app* de la aplicación.

La funcionalidad básica de **getResource** buscará y devolverá un recurso determinado con el nombre dado. Además, la función de **resourcePath()** devolverá siempre una ruta válida hasta los recursos de la aplicación (fuentes, sonidos, imágenes, etc.).

Paquete de útiles

Por último se muestra el paquete de utilidades comunes en la figura 6.9. El mismo solo contiene la clase **Clock**, un *Singleton* que representa el reloj interno de la aplicación. El mismo es capaz de determinar el tiempo que ha transcurrido entre fotogramas para calcular de forma precisa el movimiento y físicas del videojuego.

Además proporciona la posibilidad de escalar su velocidad mediante **setTimeScale** permitiendo así las simulaciones aceleradas referenciadas en el RNF-2.

Este paquete podría contener cualquier funcionalidad común y útil para varios subsistemas pero ya que las mismas están convenientemente encapsuladas en dicho subsistemas no ha sido necesario agrupar muchas funcionalidades en el presente paquete.

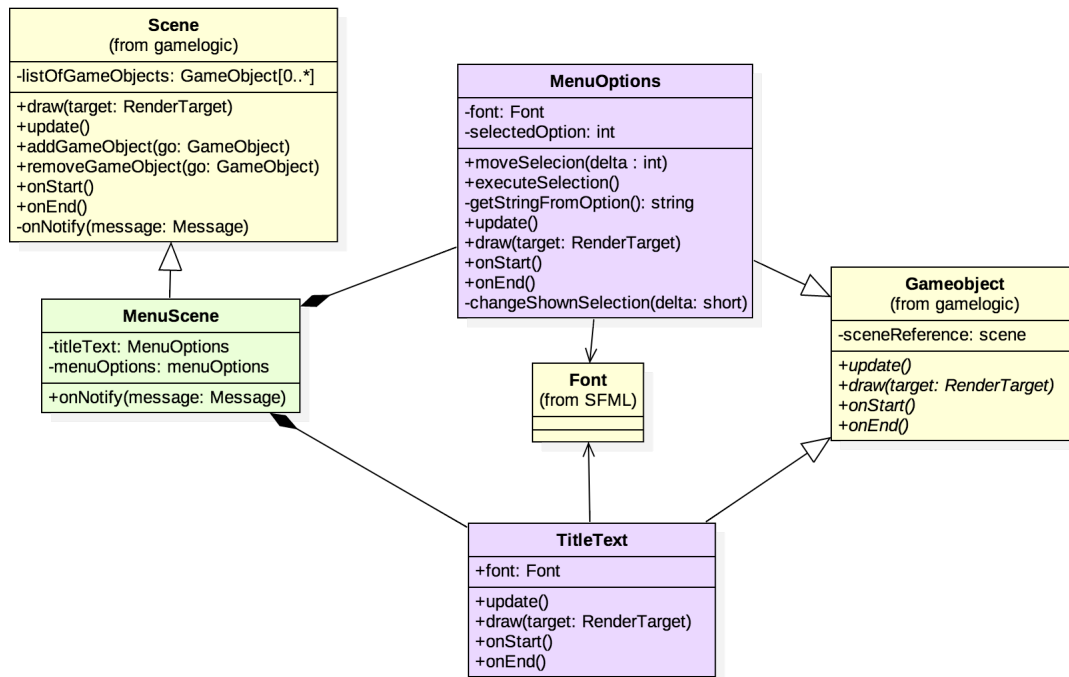


Figura 6.10: Diagrama de clases de la escena del menú

6.1.5. Escenas

Escena del menú

La escena del menú, mostrada en la figura 6.10, es la escena más sencilla de la aplicación. Contiene los siguientes *gameobjects*:

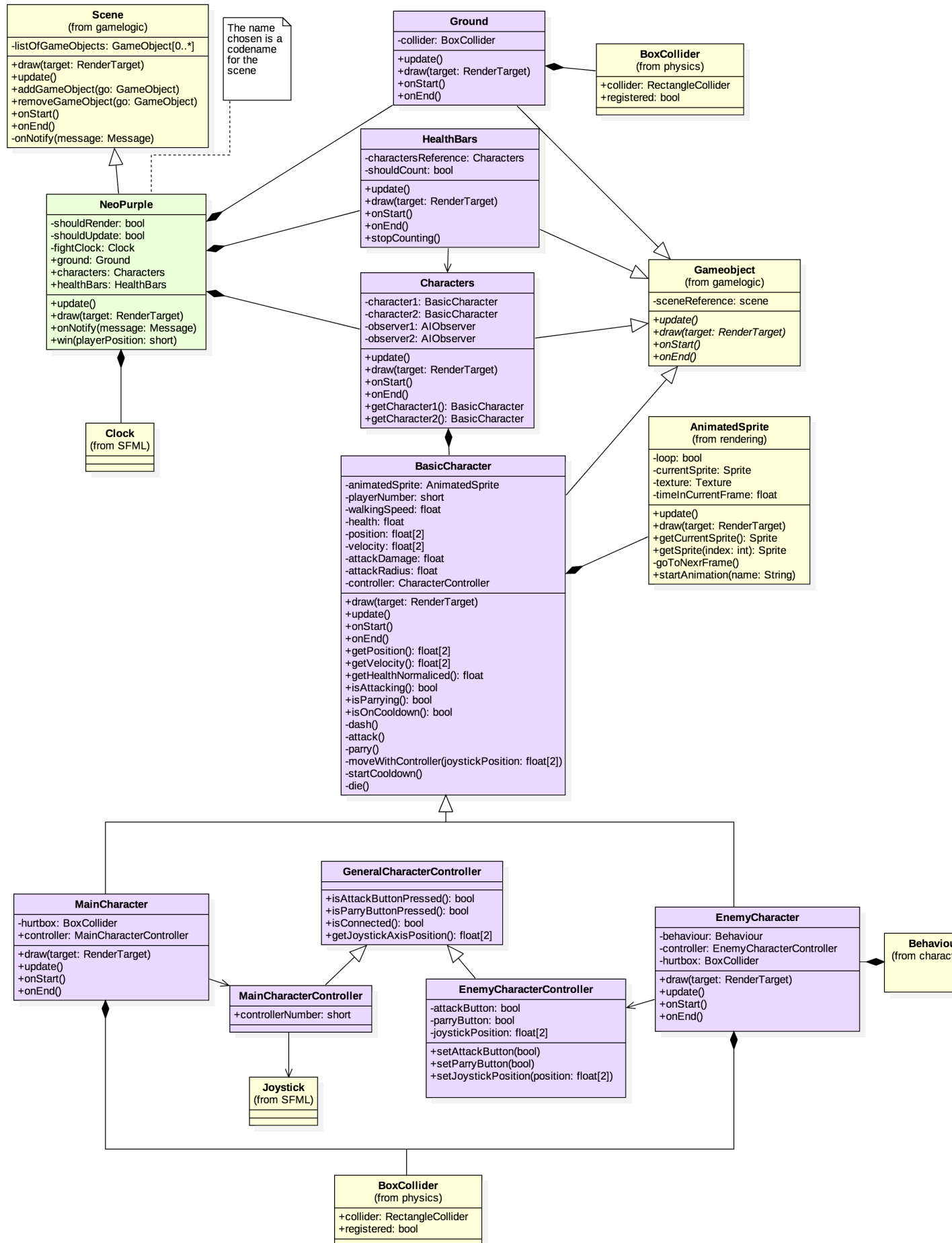
- **TitleText**: Simplemente usado para mostrar el título del juego.
- **MenuOptions**: Contiene las diferentes opciones del menú, se encarga de guardar la que está seleccionada, cambiar esta selección y ejecutar la selección cuando se requiere.

Escena de juego

Esta escena (figura 6.11) es la dedicada a jugar realmente al videojuego, la misma contiene el combate entre los dos personajes. La clase dedicada a la escena es **NeoPurple**¹ y se encarga de contener a todos los *gameobjects* necesarios y a

¹Nombre en clave escogido por la apariencia del suelo de la escena para identificarla fácilmente en caso de que se requirieran más escenas con *gameplay*

Figura 6.11: Diagrama de clases de la escena de juego



gestionar la victoria de uno de los personajes con la función **win**. Los *gameobjects* contenidos en la escena son:

- **Ground**: Encargado de representar el suelo de la escena, es el encargado tanto de pintar el suelo como de proporcionar al sistema de colisiones un *collider* que impida que los personajes se salgan de la escena mediante el uso de ese *BoxCollider*.
- **HealthBars**: Muestra la vida de los dos personajes y el tiempo de pelea restante, tiene una referencia a *Characters* para obtener información sobre la vida de ambos.
- **Characters**: Contiene e identifica a los dos personajes en la escena, no solo facilita acceder a ellos por parte de las clases que lo requieran sino que ayuda a reconocer cual es el personaje 1 y 2 y saber por que están siendo controlados (jugador, agente, sistema de reglas, etc).
- **BasicCharacter**: Clase genérica que representa un personaje, contiene todas las funciones de movimiento y acciones que el mismo puede realizar (*attack*, *parry* y *moveWithController*). Además contiene todos los atributos que definen sus características como la vida (*health*), rango (*attackRadius*), velocidad (*velocity*), posición (*position*), daño (*attackDamage*), etc.
- **MainCharacter**: Clase que representa a un personaje controlado por el jugador, hace uso de la clase **MainController** que representa un mando real conectado al equipo.
- **EnemyCharacter**: Clase que representa a un personaje controlado por la aplicación, contiene un mando virtual representado por **EnemyController** que será modificado por el comportamiento o **Behaviour** del que se habla en el siguiente apartado.

Ambos personajes cuentan con una instancia de *BoxCollider* que representa el *collider* usado para recibir daño.

Comportamiento del agente

El diagrama contenido en la figura 6.12 es uno de los más complejos de la aplicación. De hecho, debería de estar en la figura 6.11 ya que forma parte del *gameplay* pero dada su importancia y complejidad se ha decidido dedicar esta sección al mismo en aras de evitar diagramas enormes imposibles de ver en un

documento de este formato y de entender a primera vista. Para explicar sus partes se dedicará una parte deparada a cada uno de los conjuntos de componentes.

Comenzando por la parte izquierda se ven las clases que representan el estado de la pelea a ojos del agente. La versión continua del estado está contenida en *FightState* que contendrá el estado de ambos personajes representado por la clase *CharacterState*.

La clase *Observer* es el componente encargado de representar los *ojos* del agente en el sentido de ver la situación de los personajes y representarla de un modo adecuado. Por lo tanto, será este el encargado de discretizar el estado continuo para obtener un objeto de la clase *FightState Discrete*. Esta clase contiene el estado propio del personaje y del enemigo en *MyCharacterState Discrete* y *OtherCharacterState Discrete* respectivamente. Es importante mencionar que solo el estado propio contiene datos de posición ya que la misma se define de forma relativa al enemigo en la clase *Position Discrete* de forma que se evita duplicar información.

Pasando ahora a la jerarquía de clases de *Behaviour* se ve como la clase principal es la encargada de representar la estrategia genérica del patrón *Strategy*5.5 ya que el enemigo siempre tendrá una pero podrá cambiar entre ellas en diferentes ejecuciones. Esta clase *Behaviour* tiene acceso a la clase *Actions* que encapsula las acciones posibles a realizar y las ejecuta con *execute* sobre la referencia al *controller* que contiene. Algo importante es el hecho de que *Behaviour* contiene un *thread* independiente que correrá todos los cálculos referentes a la inteligencia artificial del juego, separando de forma efectiva la complejidad del juego con la del agente y favoreciendo el rendimiento.

RuleBasedBehaviour es la implementación base sobre la que probar al agente. La misma especifica las acciones que llevará a cabo el enemigo en cada situación y se ha realizado gracias al **conocimiento experto** del desarrollador pues el mismo tiene experiencia jugando al juego. Solo necesita acceso al estado actual de la pelea (*currentState*) para elegir la acción programada.

ReinforcementBehaviour es la clase que contiene el comportamiento que el agente ha aprendido. El mismo se ha generado a partir de la exploración de estados posibles y de la mejora en la función de *fitness* que ha supuesto cada acción. Para calcular este *fitness* se hace uso de la función *calculateFitness* en *Observer*.

Para representar la información referente a los estados visitados se guarda el estado actual (*currentState*) y el último (*lastState*), además de la última acción

escogida (*lastAction*). Con estos datos, al principio de cada iteración del *update*, se guardará en el *StateActionContainer* la mejora en el *fitness* asociada a una acción para un determinado estado. Luego se obtendrá el estado actual y si este no ha sido visitado se escogerá una acción aleatoria, si por el contrario este ha sido visitado, se escogerá entre las posibles acciones teniendo en cuenta el *fitness* esperado para cada una, habiendo una posibilidad de que aleatoriamente se escoja una acción con poco *fitness* para solucionar el conocido problema de exploración contra explotación² en inteligencia artificial.

Por último, *StateActionContainer* será un *Singleton* encargado de representar el conocimiento sobre los estados. Las estructuras que representan este conocimiento son *StateActionSituation* y *ActionSituation* que contienen la información para un estado y para cada acción dentro del estado respectivamente. Además, *StateActionContainer* es capaz de leer y guardar este conocimiento desde un archivo dentro del ejecutable para mantenerlo entre diferentes ejecuciones. Se podría usar la opción de *resetKnowledge* si se quisiera que el agente olvidara todo lo aprendido hasta el momento.

² Problema común en entornos de inteligencia artificial en el cual un agente debe balancear escoger siempre la opción que parece la mejor en este momento con explorar las otras posibles opciones en busca de otra que pueda ser superior.

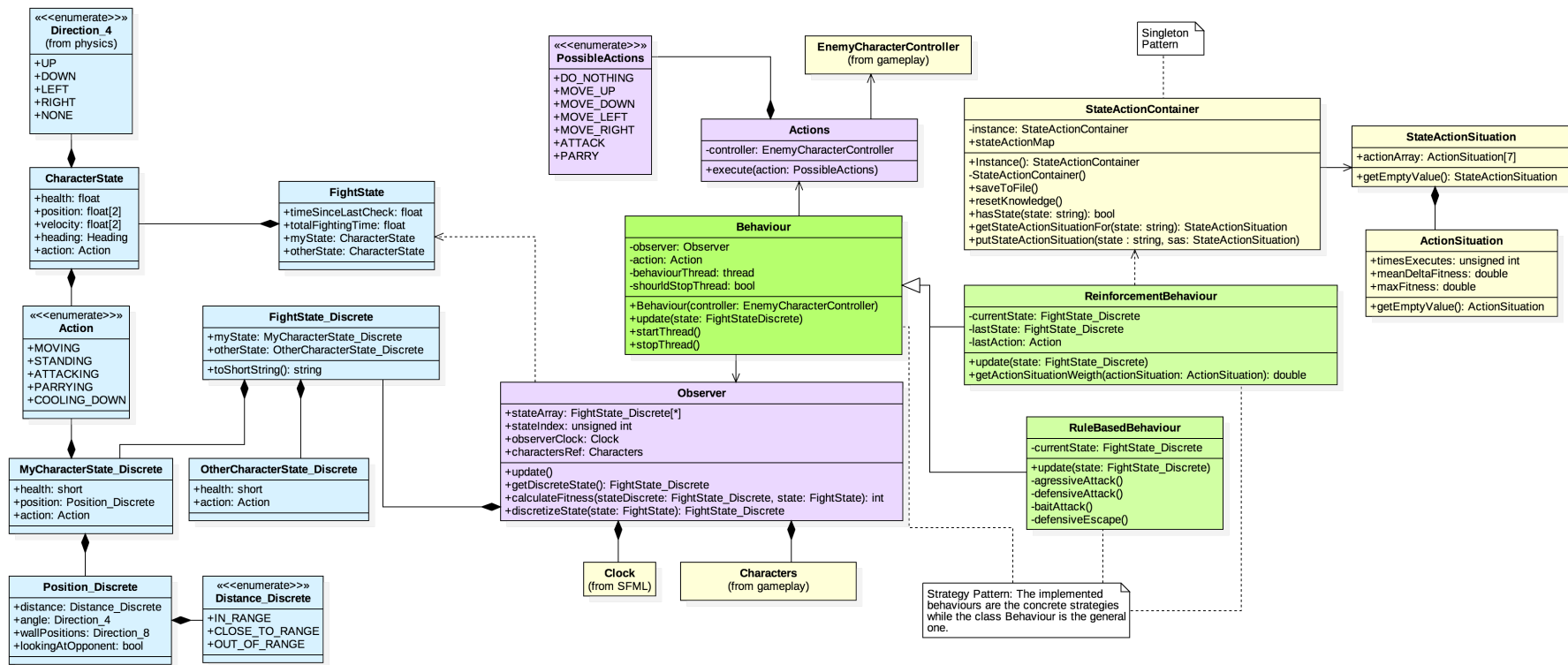


Figura 6.12: Diagrama de clases del comportamiento del agente

6.2. Diagramas de secuencia

Esta sección estará dedicada a explicar la secuencia de operaciones que se realiza en casos comunes e importantes de partes de la aplicación. Se ha optado por mostrar partes clave de la aplicación en lugar de dedicar un diagrama de secuencia a cada caso de uso ya que si se hiciera de este modo se repetiría mucha información entre los mismos, además de dar lugar a diagramas de secuencia de tamaños excesivos. Otra nota importante es que no se suelen utilizar diagramas de secuencia en este tipo de arquitecturas y en especial en videojuegos ya que su naturaleza lleva a realizar acciones durante muchas diferentes del *gameloop*³ haciendo que representar un caso de uso diera lugar a diagramas de secuencia con decenas de instancias y cientos de mensajes entre ellas.

En los siguientes párrafos se mostrarán diagramas de secuencia relevantes de la aplicación y se relacionarán con los casos de uso en los que son importantes. Se empieza por los diagramas asociados a la aplicación en general para luego seguir con los diagramas específicos relacionados con las escenas y el agente.

6.2.1. Aplicación general

Aplicación global

La figura 6.13 muestra la creación y destrucción de los subsistemas de la aplicación, además del bucle principal del juego conocido como *gameloop*. El diagrama en si mismo es sencillo, se crean todos los subsistemas y finalmente se destruyen en orden inverso.

El bucle contiene las actualizaciones de todas las partes de la aplicación en el mismo orden en el que se han creado aunque este podría variar. Esto podría dar lugar a pensar que el hecho de realizar el bucle en este orden de problemas en el sentido de que si un subsistema quiere modificar otro que ya ha sido actualizado, dicha modificación tiene que esperar a la siguiente iteración. De forma efectiva esto no es un problema dado que al realizar el intercambio de mensajes al final es el bus de mensajes el encargado de hacer que se ejecuten todas las funciones relacionadas con los mensajes existentes. Como mucho es posible que en algunas situaciones la ejecución de una funcionalidad se retrase una iteración pero dado que cada fotograma (y por lo tanto, cada iteración) se muestra cada unos 16 milisegundos⁴ no es posible que el usuario sea consciente.

³Bucle principal de un videojuego en el que se actualizan todas sus partes, se puede ver un ejemplo en la figura 6.13

⁴contando con una frecuencia de actualización común de 60 fotogramas por segundo, que se ha probado como constante en esta aplicación en equipos relativamente actuales

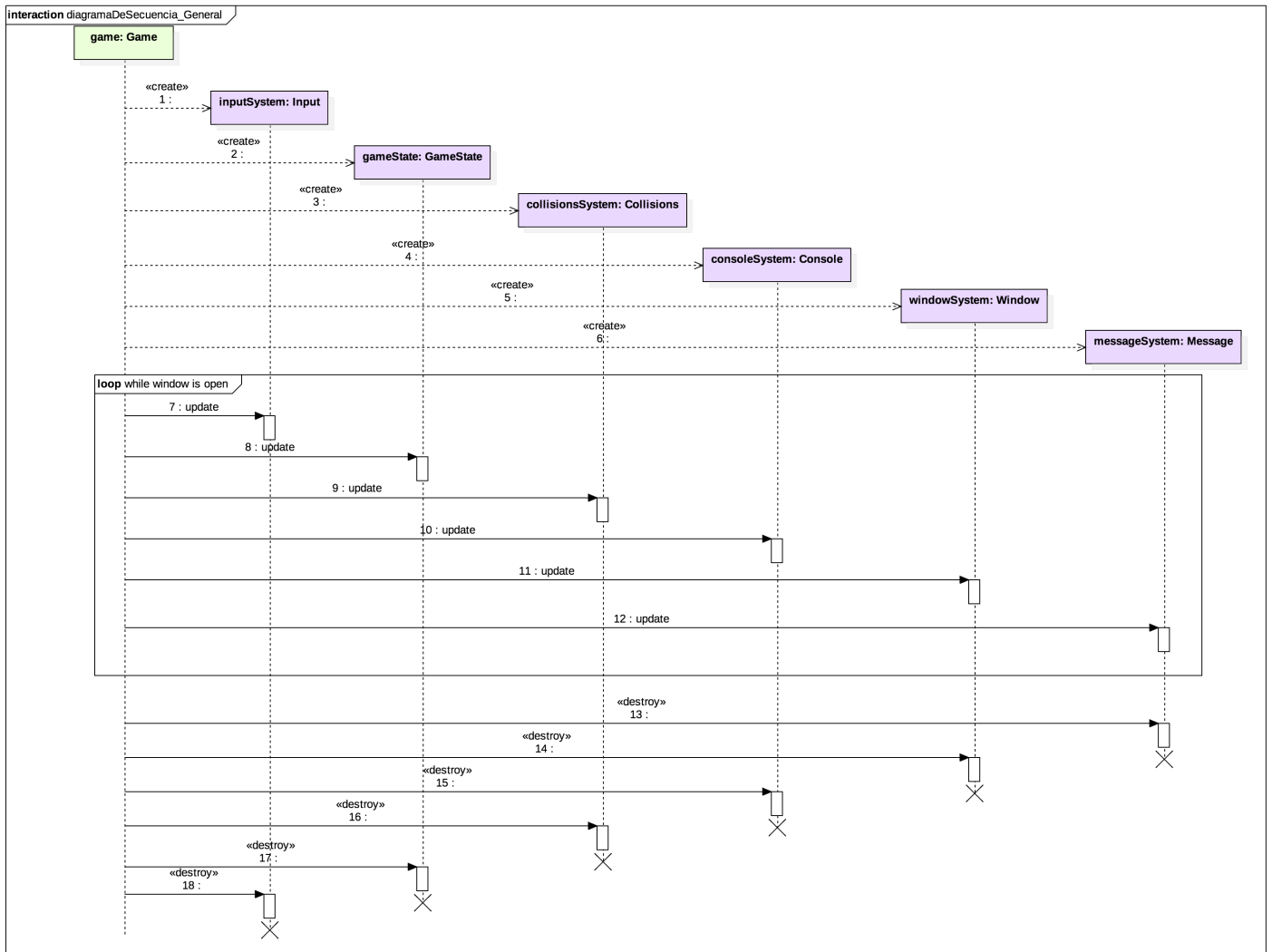


Figura 6.13: Diagrama de secuencia general de la aplicación

Este diagrama se relaciona estrechamente con los casos de uso CU-1 (iniciar aplicación) y CU-2 (cerrar aplicación).

Escena genérica

El diagrama de secuencia de la figura 6.14 nos muestra como el objeto *gameState* puede crear y destruir una escena para cambiar entre las mismas, lo que se relaciona estrechamente con el CU-3 (seleccionar y ejecutar opción). Además se muestra como es el *gameState* el encargado de actualizar la escena pero será el subsistema de *rendering* con su clase *Window* el encargado de hacer que se dibuje la escena y todos sus *gameobjects*.

Consola

Relacionado estrechamente con el caso de uso CU-9 (Visualizar resultados de combates) y el requisito no funcional RNF-4 (Facilidad para depurar) está el diagrama de secuencia mostrado en la figura 6.15.

En orden temporal, se muestra como el usuario puede introducir caracteres por teclado para crear un comando en la consola para luego ejecutarlo y que el mismo se envíe al bus de mensajes. Luego se observa como la clase *Window*, al tener acceso a la consola, es capaz de comprobar si está abierta y dibujarla por pantalla junto con todas las líneas de texto que esta contiene.

Finalmente se muestra como el usuario puede alternar entre visualizar o no la consola al pulsar una determinada tecla interpretada por el sistema de *input*.

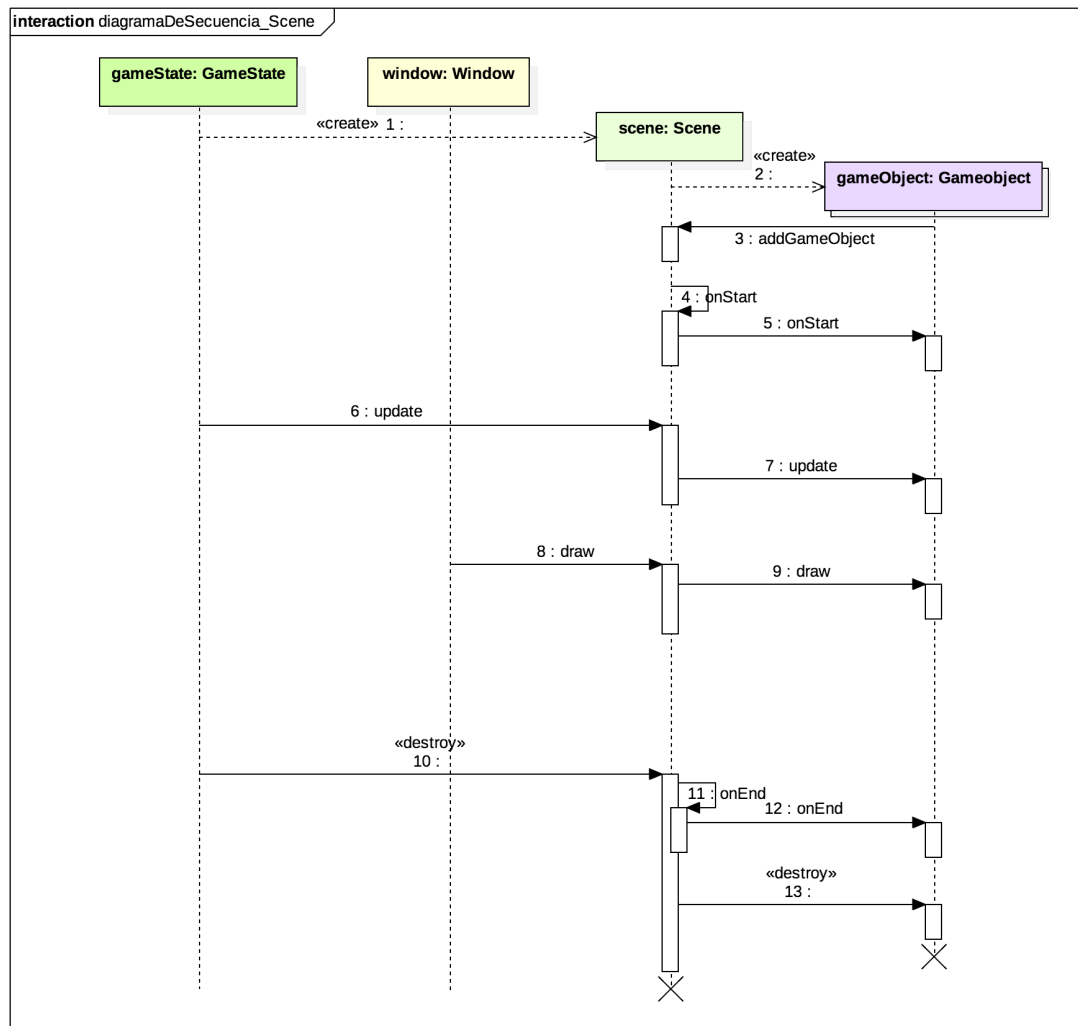


Figura 6.14: Diagrama de secuencia genérico de una escena

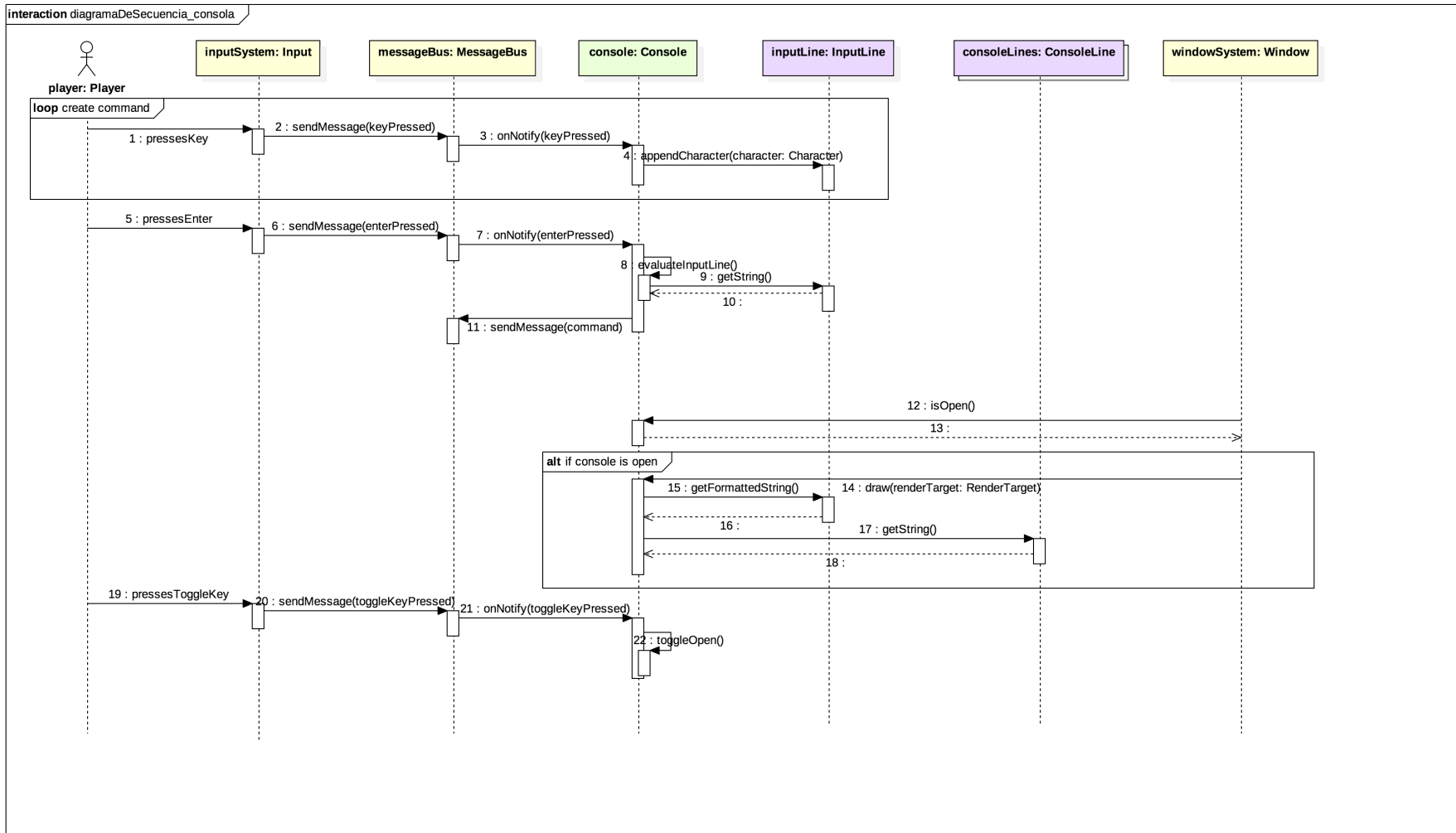


Figura 6.15: Diagrama de secuencia de la consola

6.2.2. Escenas y agente

Escena del menú

En el diagrama de secuencia de la escena del menú de la figura 6.16 se ve como el usuario es capaz de seleccionar entre las opciones del menú para luego ejecutar la seleccionada y cambiar de escena.

El flujo comienza por utilizar el sistema de input para mandar mensajes referentes a cambiar la opción seleccionada. Una vez que se está en la opción deseada se envía el mensaje de ejecutar dicha opción. En este momento se envía un mensaje que llega al *gameState* indicándole la nueva escena a la que habrá que cambiar, en este momento el *gameState* destruye la escena del menú y genera la nueva escena según la opción seleccionada.

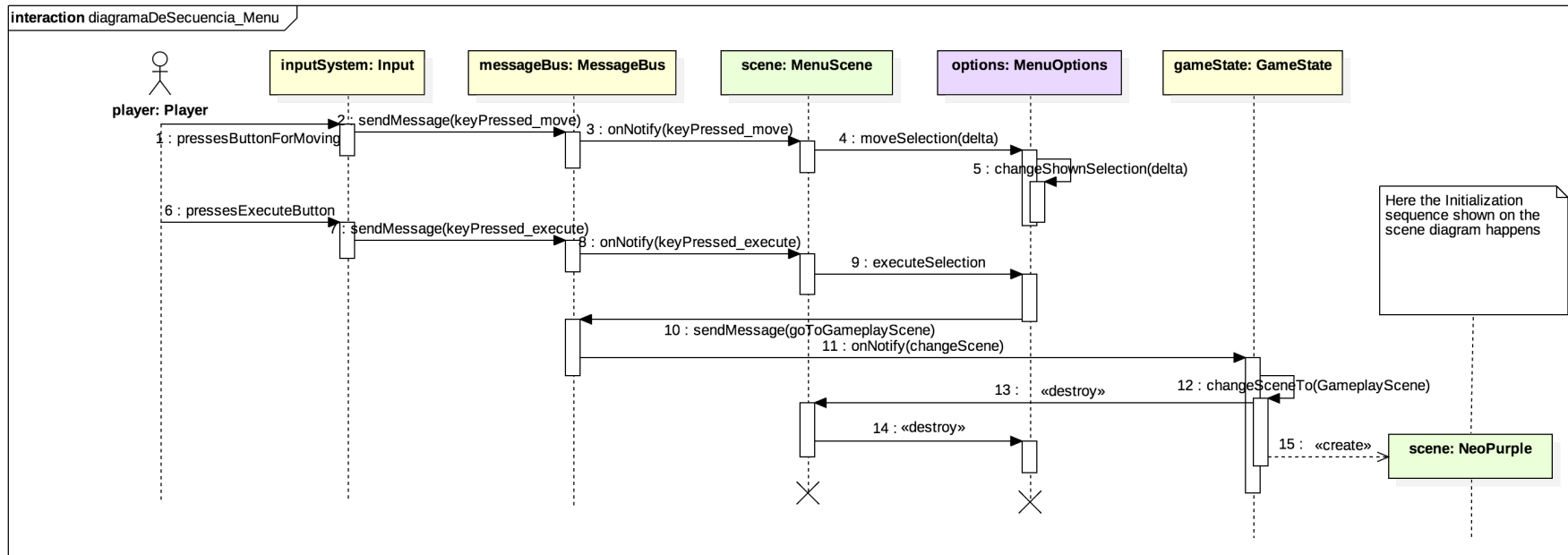


Figura 6.16: Diagrama de secuencia de la escena del menú

Escena del juego

La figura 6.17 contiene el diagrama de secuencia que indica la interacción de un jugador con el videojuego, relacionado con los casos de uso CU-7 (jugador contra agente) y CU-8 (jugador contra jugador).

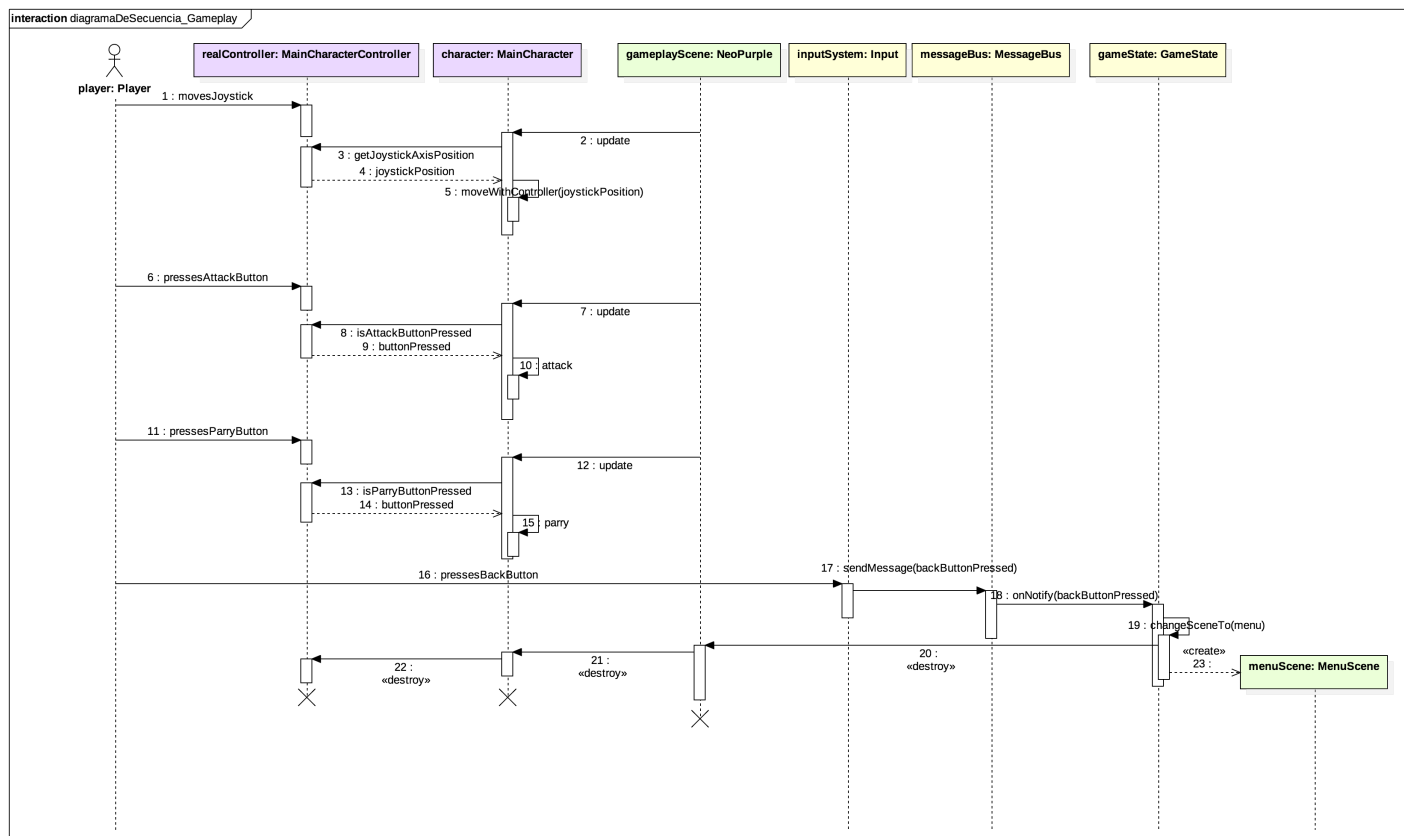
En el diagrama se muestra como el jugador realiza todas las acciones disponibles para el durante un combate en el siguiente orden temporal:

1. El jugador mueve el *joystick* del mando y el mismo se traduce en su personaje moviéndose por la escena.
2. El jugador pulsa el botón de atacar en el mando y su personaje realiza un ataque.
3. El jugador pulsa el botón de defenderse y su personaje hace lo propio.
4. El jugador pulsa el botón de volver al menú, el combate y la escena terminan y se vuelve a la escena del menú.

Agente

Finalmente, la figura 6.18 contiene el diagrama de secuencia asociado al agente y sus acciones. Será más sencillo comprenderlo al relacionarlo con el diagrama de clases del agente mostrado en la figura 6.12. Para explicar la secuencia de pasos que se están dando se hace uso de la siguiente enumeración:

1. La escena actualiza al agente y este hace lo mismo con su *Observer*.
2. El comportamiento (*Behaviour*) se actualiza y discretiza el estado continuo mediante el uso del *Observer*.
3. Se actualiza el conocimiento del agente enviando los datos al *StateAction-Container* y se obtiene del mismo el conocimiento previo sobre el estado actual.
4. Se calcula la acción a llevar a cabo y se ejecuta con la ayuda de *Actions*. Este objeto modificará el *controller* con los parametros adecuados.
5. El personaje leerá los datos del *controller* y ejecutará una de las acciones según el estado de este mando virtual.

Figura 6.17: Diagrama de secuencia de la escena del *gameplay*

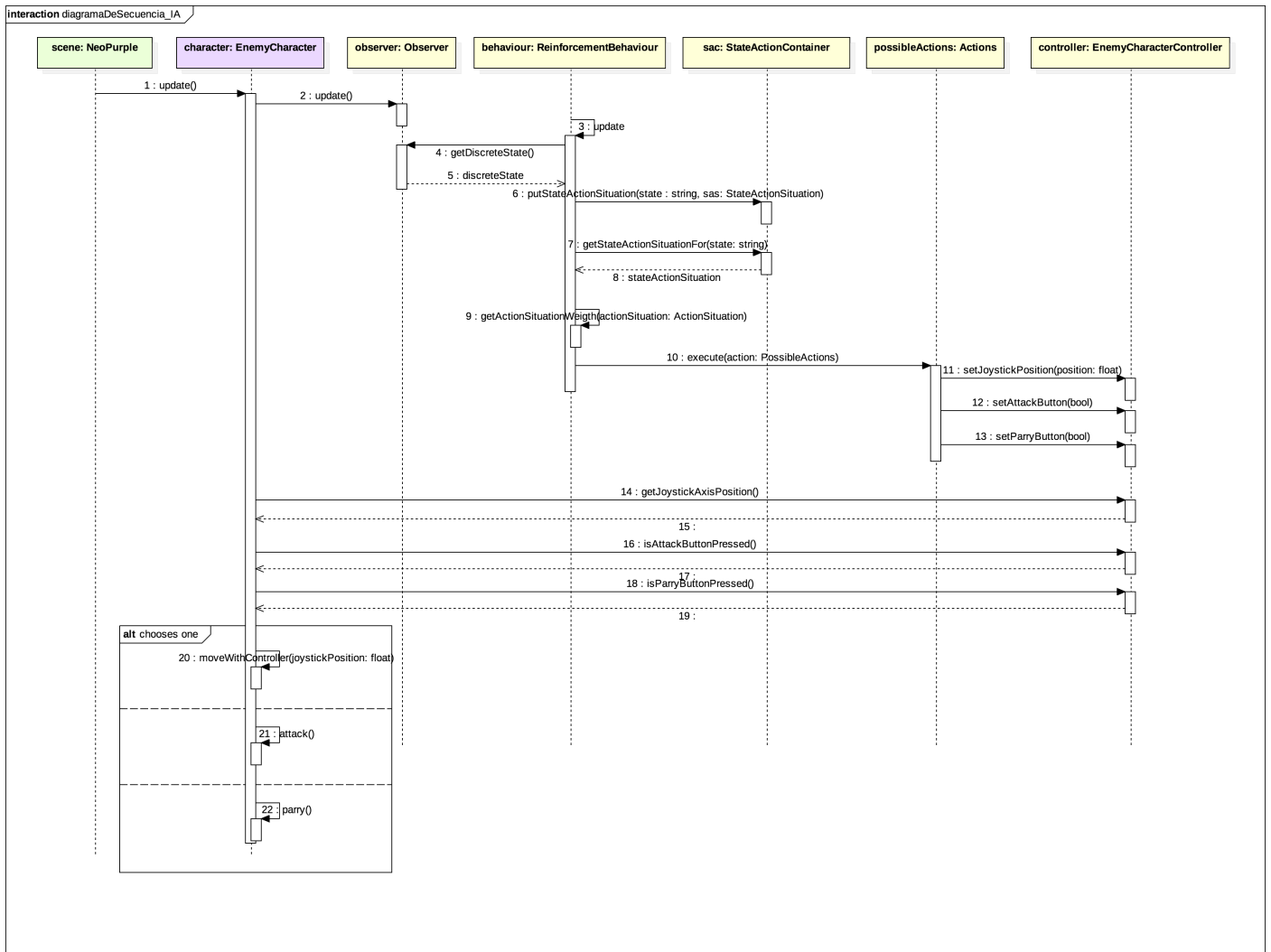


Figura 6.18: Diagrama de secuencia del agente

6.3. Diseño de la interfaz gráfica

Este apartado muestra los diseños realizados primero en papel en referencia a la interfaz gráfica de la aplicación. Dichos diseños son sencillos ya que a diferencia de aplicaciones de otros tipos, los videojuegos de este género suelen optar por evitar una gran cantidad de opciones en una misma vista tendiendo normalmente al minimalismo.

La ventana tendrá una relación de aspecto constante de 4:3 para evocar un parecido con videojuegos antiguos que utilizaban una relación similar dados los estándares de los televisores de la época. Esto debería favorecer la comodidad y dar un aspecto general familiar a los usuarios.

En relación con el requisito no funcional RNF-8 (referente a la usabilidad de la aplicación) se ha buscado la máxima comodidad a la hora de controlar el juego mediante el uso de un mando. Por ello, todos los controles, tanto del menú como del combate están enfocados a ser realizados con un mando similar al de una videoconsola, dejando a un lado el uso del teclado solamente para funciones de depuración o avanzadas tales como abrir y escribir en la consola.

Otro aspecto muy relacionado con la usabilidad es el uso de un indicador sonoro a todas las acciones importantes que se realizan. Dicho indicador tiene que ser constante con la acción que se está mostrando en la interfaz.

6.3.1. Interfaz del menú

En el *mockup*⁵ del menú mostrado en la figura 6.19 se vé como se ha echo hincapié en la simplicidad del mismo. El título se muestra utilizando un tamaño de fuente superior y las opciones se alinean por la izquierda teniendo en cuenta que pueden tener una longitud distinta.

La opción seleccionada se muestra en negrita y subrayada para no dar lugar a dudas a la hora de distinguirla de las otras. Tanto el cambio entre opciones como la ejecución de una de ellas tienen un sonido asociado lo suficientemente diferente como para ser inconfundibles entre ellos.

A la hora de colocar el título y las opciones se ha utilizado la conocida como regla de los tercios⁶

⁵modelo de diseño de una interfaz que permite realizar cambios a la misma sin necesidad de implementarla y con el fin de evaluar a grandes rasgos su usabilidad y aspecto

⁶Regla, generalmente asociada a la fotografía, que sugiere que alinear vertical y horizontalmente los elementos de una fotografía con las líneas imaginarias que separa la imagen en tercios las hace más interesantes, naturales y cómodas a la vista

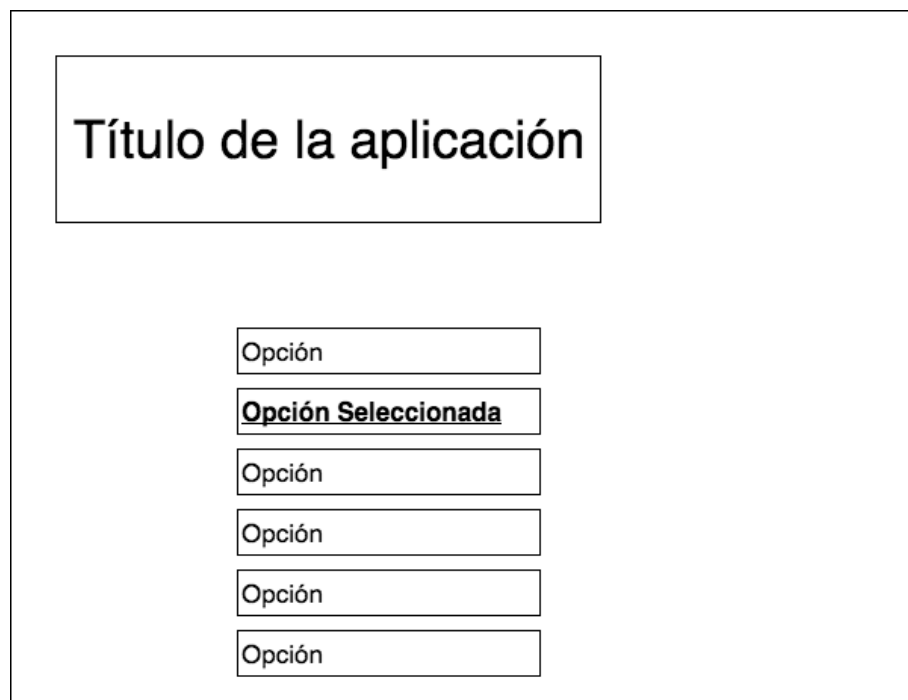


Figura 6.19: *Mockup* de la interfaz del menú

6.3.2. Interfaz del combate

La figura 6.20 contiene el *mockup* de la interfaz que el usuario ve al realizar el combate entre personajes. En el mismo se distinguen dos partes diferenciadas:

En la parte superior de la pantalla se ven las barras de vida de los personajes identificadas por colores, dichas barras de vida decrecerán en tamaño horizontalmente para indicar que el personaje asociado ha recibido daño. En el centro se ve un número que indica los segundos restantes del combate, cuando dicho contador llega a cero se termina el combate y gana el jugador con más vida, empatando si es igual. Este tipo de interfaz de combate pretende recordar a la empleada por antiguos juegos de peleas 2D en recreativas tales como *Street Fighter* o *Mortal Kombat* lo que ayuda a que el usuario la considere familiar.

En el centro de la pantalla se puede ver claramente la zona de combate con ambos personajes. Dicha zona se diferencia del resto de la pantalla al utilizar un fondo completamente negro haciendo sencillo el hecho de darse cuenta de que la zona en la que se pueden mover los personajes está limitada. Además los personajes cuentan con un indicador por colores que los relacionan con sus barras de vida para facilitar su identificación durante la batalla.



Figura 6.20: *Mockup* de la interfaz del combate

Durante el combate el hecho de atacar producirá un sonido específico, de la misma forma que recibir daño. Esto incrementa la familiarización del jugador ya que combina la animación mostrada y el sonido con la acción que acaba de ocurrir.

Pese a que no se muestre en la figura, cuando se detiene el combate porque uno de los personajes ha derrotado al otro se muestran letras que indican el jugador ganador, ocupando las mismas toda la pantalla y siendo dibujadas con el mismo color que identifica a cada personaje.

6.3.3. Interfaz de la consola

Finalmente, la figura 6.21 muestra como se vería la consola sobre la aplicación. Recordemos que la misma se puede abrir y cerrar en cualquier escena por lo que tiene un fondo coloreado pero que cierta transparencia que permita ver fácilmente lo que está ocurriendo.

Al fondo de la consola se puede observar una línea con el símbolo > que identifica a la línea de entrada para el usuario. SI la consola está abierta y se escribe en el teclado los caracteres se agregarán a esta línea, siendo limpiada al

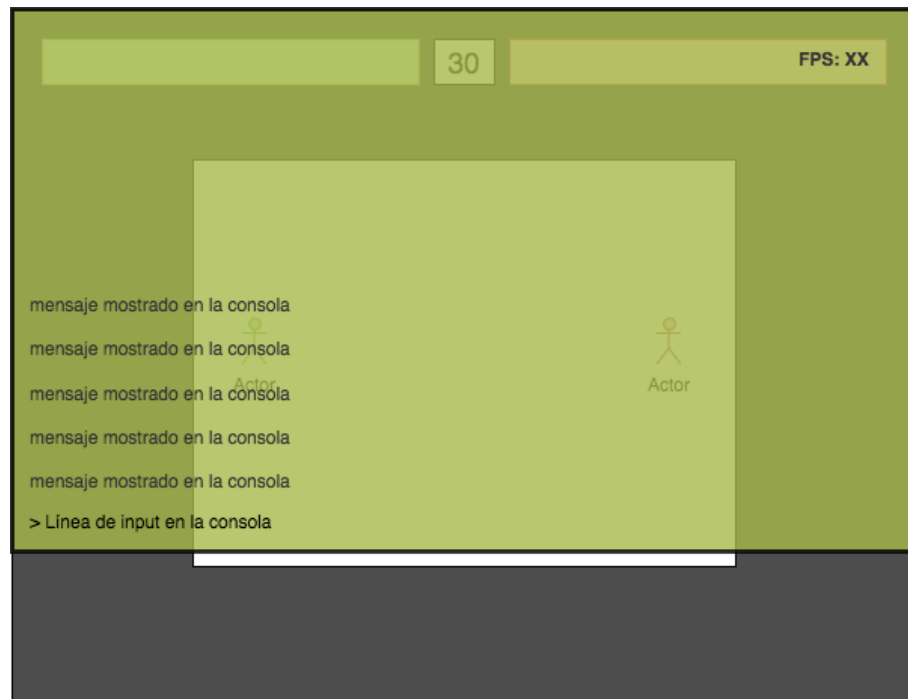


Figura 6.21: *Mockup* de la interfaz de la consola

pulsar *intro*. Las líneas en un negro ligeramente más suave situadas por encima representan los mensajes relevantes que la consola está mostrando al usuario. Finalmente, en la esquina superior izquierda se puede ver un contador de fotogramas por segundo o *FPS* como siglas en inglés de *Frames Per Second*.


Capítulo 7

Validación y pruebas


Completar
Valida-
ción y
pruebas

Capítulo 8

Valoraciones finales



Completar
Valora-
ciones
finales



Quitar
la sec-
ción de
ejemplos

Capítulo 9

Exemplos

9.1. Un exemplo de sección

Esta é *letra cursiva*, esta é **letra negrilla**, esta é letra subrallada, e esta é letra curier. Letra tiny, scriptsize, small, large, Large, LARGE e moitas más. Exemplo de fórmula: $a = \int_0^\infty f(t)dt$. E agora unha ecuación aparte:

$$S = \sum_{i=0}^{N-1} a_i^2. \quad (9.1)$$

As ecuaciones se poden referenciar: ecuación (9.1).

9.1.1. Un exemplo de subsección

O texto vai aquí.

9.1.2. Outro exemplo de subsección

O texto vai aquí.

9.1.2.1. Un exemplo de subsubsección

O texto vai aquí.

9.1.2.2. Un exemplo de subsubsección

O texto vai aquí.

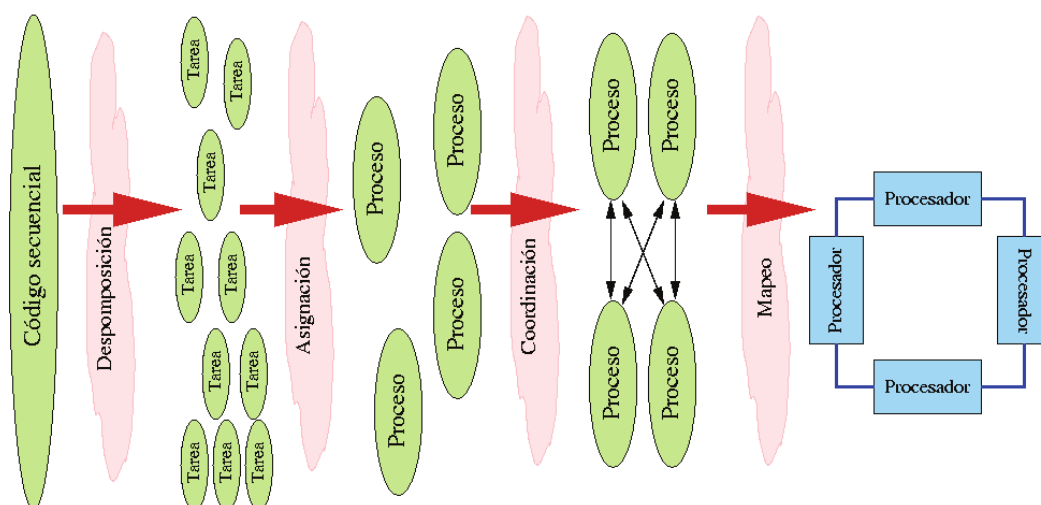


Figura 9.1: Esta é a figura de tal e cal.

Izquierda	Derecha	Centrado
ll	r	cccc
llll	rrr	c

Cuadro 9.1: Esta é a táboa de tal e cal.

9.1.2.3. Un exemplo de subsubsección

O texto vai aquí.

9.2. Exemplos de figuras e cadros

A figura número 9.1.

O cadro (taboa) número 9.1.

9.3. Exemplos de referencias á bibliografía

Este é un exemplo de referencia a un documento descargado da web [10]. E este é un exemplo de referencia a unha páxina da wikipedia [11]. Agora un libro

[12] e agora unha referencia a un artigo dunha revista [13]. Tamén se poden pór varias referencias á vez [10, 12].

9.4. Exemplos de enumeracións

Con puntos:

- Un.
- Dous.
- Tres.

Con números:

1. Catro.
2. Cinco.
3. Seis.

Exemplo de texto verbatim:

```
0 texto          verbatim
  se visualiza tal
    como se escribe
```

Exemplo de código C:

```
#include <math.h>
main ()
{   int i, j, a[10];
    for(i=0;i<=10;i++) a[i]=i; // comentario 1
    if(a[1]==0) j=1; /* comentario 2 */
    else j=2;
}
```

Exemplo de código Java:

```
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello_World!"); // Display the string.
    }
}
```


Capítulo 10

Conclusiones y posibles ampliaciones

Conclusiones e posibles ampliaciones

Completar
conclu-
siones y
posibles
amplia-
ciones

Apéndice A

Manuais técnicos

Manuais técnicos: en función do tipo de Traballo e metodoloxía empregada, o contido poderase dividir en varios documentos. En todo caso, neles incluírase toda a información precisa para aquelas persoas que se vaian a encargar do desenvolvemento e/ou modificación do Sistema (por exemplo código fonte, recursos necesarios, operacións necesarias para modificacións e probas, posibles problemas, etc.). O código fonte poderase entregar en soporte informático en formatos PDF ou postscript.

Realizar
manual
técnico

Apéndice B

Manuais de usuario

Manuais de usuario: incluírán toda a información precisa para aquelas persoas que utilicen o Sistema: instalación, utilización, configuración, mensaxes de erro, etc. A documentación do usuario debe ser autocontida, é dicir, para o seu entendemento o usuario final non debe precisar da lectura de outro manual técnico.

Realizar
manual
de usua-
rio

Apéndice C

Licencia

Insertar
licencia
apropia-
da

Completar
biblio-
grafía

Bibliografía

- [1] PMI. Guía de los fundamentos para la dirección de proyectos (Guía del PMBOK), Quinta edición. PMI, 2013.
- [2] Definición de arquitectura según la ISO/IEC/IEEE 42010:2011. Artículo referenciado (<http://www.iso-architecture.org/ieee-1471/defining-architecture.html>). Consultado el 2 de junio de 2017.
- [3] Grady Booch, James Rumbaugh, and Ivar Jacobson. El lenguaje unificado de modelado. Manual de referencia. Addison Wesley, 1999.
- [4] IEEE recommended practice for software requirements specifications. Technical report, 1998.
- [5] Nielsen, J., and Molich, R. Heuristic evaluation of user interfaces, Proc. ACM CHI'90 Conf. (Seattle, WA, 1-5 April), 249-256, 1990.
- [6] INTECO. Guía práctica de gestión de configuración LNCS. INTECO, 2008.
- [7] The Standish Group. Chaos Report. The Standish Group, 2016.
- [8] Vitae. Estudio salarial 2015-2016. Vitae Consultores, 2016.
- [9] Jason Gregory. Game Engine Architecture, Second Edition 2nd. A. K. Peters, Ltd. Natick, MA, USA, 2014.
- [10] Nvidia CUDA programming guide. Versión 2.0, 2010. Disponible en <http://www.nvidia.com>.
- [11] Acceso múltiple por división de código. Artigo da wikipedia (<http://es.wikipedia.org>). Consultado o 2 de xaneiro do 2010.
- [12] R.C. Gonzalez e R.E. Woods, *Digital image processing*, 3ª edición, Prentice Hall, New York, 2007.

- [13] P. González, J.C. Cartex e T.F. Pellas, “Parallel computation of wavelet transforms using the lifting scheme”, *Journal of Supercomputing*, vol. 18, no. 4, pp. 141-152, junio 2001.

Quitar
esta lista
de TO-
DOs

Todo list

Completar la organización del documento	3
Completar Validación y pruebas	117
Completar Valoraciones finales	119
Quitar la sección de ejemplos	121
Completar conclusiones y posibles ampliaciones	127
Realizar manual técnico	129
Realizar manual de usuario	131
Insertar licencia apropiada	133
Completar bibliografía	135
Quitar esta lista de TODOs	139

