Treball final de grau

Grau en Disseny i Desenvolupament de Videojocs

Aplicación móvil Android de gestión semi-automática de juego de rol.

Document: Memoria

Alumne: Èric Galán Sola

Tutor: Gustavo Ariel Patow
Departament: Informàtica i Matemàtica Aplicada
Àrea: Llenguatges i Sistemes Informàtics

Convocatòria Juny 2023

1. Introducción, motivaciones, propósito y objetivos del proyecto y distribución de tareas 7
1.1. Introducción 7
1.2. Motivaciones 8
1.3. Propósito y objetivos del proyecto 9
1.4. Distribución de tareas 9
2. Estudio de viabilidad 11
2.1. Viabilidad y recursos 11
2.1.1. Recursos de hardware 11
2.1.2. Recursos de software 12
2.1.3. Recursos humanos 12
2.1.4. Viabilidad económica 13
2.4. Estudio de mercado 14
2.4.1. Estado del arte 14
2.4.2. Comparación del estado del arte 16
2.4.3. Modelo de negocio 17
2.5. Público objetivo y perfil de jugador 19
3. Planificación 21
3.1. Tareas planificadas 21
3.2. Organización de las tareas 22
3.3. Metodología de trabajo 23
4. Marco de trabajo y conceptos previos 25
4.1. Conceptos previos 25
4.2. Entornos de trabajo 27
4.2.1. Entorno de implementación 27
4.2.2. Entornos de diseño 28
4.2.3. Entornos de documentación 30
4.2.4. Entornos de planificación y gestión 30
5. Diseño del proyecto 33
5.1. Glosario del diseño 33
5.2. Mecánicas 33
5.2.1. Flujo de juego 34

5.2.2. Personajes 36
5.2.2.1. Creación de personajes 37
5.2.3. Valores 40
5.2.3.1. Bonos de valor 41
5.2.3.2. Valores físicos 42
5.2.3.3. Valores almáticos 43
5.2.3.4. Dados de ataque 45
5.2.3.5. Acciones por turno 45
5.2.4. Razas 45
5.2.5. Clases 47
5.2.6. Equipo 50
5.2.6.1. Armas 50
5.2.6.2. Objetos 53
5.2.6.3. Monedero y la economía monetaria 55
5.2.7. Habilidades 56
5.2.8. Progresión 58
5.2.9. Acciones 61
5.2.9.1. Dificultades 66
5.2.9.2. Pifias y milagros 68
5.2.9.3. Acciones ventaja 68
5.2.9.1. Interacciones 68
5.2.9.2. Comer y Beber 70
5.2.9.3. Descanso 72
5.2.10. Combate 73
5.2.10.1. Movimiento 78
5.2.10.2. Ofensiva. 79
5.2.10.3. Estados “Inconsciente”, “Exánime” y “fuera de combate” 85
5.2.10.4. Curaciones en combate 87
5.2.11. Características del personaje 88
5.2.12. Recursos y economía interna 90
5.3. Interfaces 92
5.3.1. Menú Principal 94
5.3.2. Creador de personajes 97

5.3.3. Gestor de personaje 109
5.4. Narrativa 124
5.4.1. Sinopsis 125
5.4.2. Trasfondo del mundo 125
5.4.3. Narrative devices 127
5.4.4. Tensión dramática y jugable 127
5.4.5. Puntos de la trama, granularidad y disparadores 127
5.4.6. Estructura narrativa 127
5.4.7. Dimensión física 128
5.4.7. Dimensión temporal 128
5.4.8. Dimensión ambiental y referentes estéticos 129
5.4.9. Dimensión emocional 129
5.4.10. Aspectos éticos 129
5.5. Personajes 130
5.5.1. Razas 131
5.5.2. Clases 133
5.6. Elementos de feedback 135
5.7. Assets usados 136
5.8. Guardado y borrado de personajes 142
6. Implementación y pruebas 144
6.1. Sistema de clases del proyecto 144
6.1.1. Personaje 145
6.1.2. Data Persistance Manager 152
6.1.3. ValoresCuerpo 162
6.1.4. ValoresAlma 168
6.1.5. OtrosValores 175
6.1.6. Raza 176
6.1.7. Clase 177
6.1.8. Hechizo 178
6.1.9. Monedero 179
6.1.10 Arma 179
6.2. Implementación del sistema de carga, guardado y borrado de personajes 180

6.2.1. Singleton 181
6.2.2. Carga, guardado y borrado de personajes 182
6.3. Implementación del menú inicial 187
6.3.1. Selector de personajes 189
6.3.2. Funcionalidad de los botones 195
6.4. Implementación del creador de personajes 200
6.4.1. Navegación y organización 201
6.4.2. Asignación de valores 206
6.4.3. Selección de raza 216
6.4.4. Selección de clase 234
6.4.5. Selección del arma 241
6.4.6. Asignación de acciones ventaja 246
6.4.7. Paso final 253
6.5. Implementación del gestor de personaje 255
6.5.1. Navegación y organización 255
6.5.2. Sistema de progresión 260
6.5.3. Información básica del personaje 271
6.5.4. Valores 275
6.5.5. Acciones 292
6.5.6. Ataque, esquiva y bloqueo 309
6.5.7. Habilidades 325
6.5.8. Equipo 349
6.5.9. Características 359
6.5.10. Manual del sistema 362
6.6. Pruebas 362
7. Resultados **364**
7.1. Legislación y normativas vigentes 364
7.2. Resultado final 364
8. Conclusiones **366**
8.1. Valoración final 366
8.2. Desviaciones de la planificación original 367
9. Trabajo a futuro **370**
10. Bibliografía **371**

11. Anexos **372**
12. Manual del usuario e instalación **373**
12.1. Guía de instalación 373
12.2. Manual de usuario de la aplicación 376

1. Introducción, motivaciones, propósito y objetivos del proyecto y distribución de tareas

1.1. Introducción

El afán de encarnar a un personaje y vivir sus aventuras es uno de los aspectos más básicos de los videojuegos, pero esta idea no nace en este pasatiempo, sino de los juegos de rol de mesa. De hecho, los juegos de rol de mesa han servido de inspiración para crear varios géneros y sagas enteras de videojuegos de gran popularidad, como Fallout o Final Fantasy.

En resumidas cuentas, un juego de rol puede definirse como un juego donde uno o varios jugadores interpretan el papel de un personaje, encarnando un rol o personalidad determinados. A esto se le llama "rolear" o "role playing". Los jugadores proyectan las personalidades y roles de los personajes que controlan durante una historia o campaña en la que deben superar desafíos de diversa índole desde el punto de vista de sus personajes, improvisando sobre la marcha.

Normalmente, para ayudar a los jugadores a interpretar las diversas situaciones que ocurren durante las partidas, se sigue un conjunto determinado de reglas que sirven para definir los resultados de dichas situaciones, según las aptitudes de los personajes. A este conjunto de reglas se le llama "sistema".

El mayor referente de los sistemas de los juegos de rol actualmente es Dungeons & Dragons, o Dragones y Mazmorras en español. Este sistema es considerado el más asequible y sencillo, pero aún así, presenta un problema para los jugadores noveles, su curva de aprendizaje. Los jugadores nuevos suelen intimidarse ante la cantidad de información que deben aprender para entender cómo crear su personaje y como jugar al juego. Actualmente hay soluciones parciales a ese problema, como ofrecer personajes ya creados para poder interpretarlos y campañas preestablecidas a las que aventurarse, pero esto hace perder una parte importante de la gracia de los juegos de rol, crear una historia propia con sus propios personajes. Por otra parte, la naturaleza compleja de los juegos de rol abruma a los jugadores noveles por la extraordinaria cantidad de información que se debe tener en cuenta en todo momento, lo cual llega a ser

contraproducente para la experiencia de estos jugadores. Y aunque ya existan herramientas que asisten a los jugadores en ciertos elementos de las partidas, a causa de la inherente complejidad de los sistemas, no llegan a ser muy intuitivas para jugadores nuevos.

Todas estas observaciones sobre la relación de los jugadores noveles con la complejidad de los juegos de mesa de rol se aprovecharán en este proyecto para ofrecer una alternativa que facilite una entrada fácil y cómoda a los juegos de rol puros. En este caso, se desarrollará una aplicación móvil Android que ofrece herramientas fáciles de entender para facilitar las partidas de rol de los jugadores novatos junto a un sistema de rol simple, llamado Animaia, en el que se basará la aplicación.

1.2. Motivaciones

Vistas las observaciones del punto anterior, la motivación principal de este proyecto es proponer una alternativa a los juegos de rol de mesa, una puerta de entrada más accesible para los jugadores nuevos a este pasatiempo, donde poder disfrutar de la experiencia del roleplay sin tener que preocuparse de las complejidades del sistema que soporta la partida.

Este proyecto ofrece la oportunidad de cumplir las motivaciones personales siguientes:

- Hacer uso de todos los conocimientos adquiridos durante el grado para dar forma a este proyecto.
- Poder aprender el proceso de creación de una aplicación móvil, y todas las características de desarrollo que eso conlleva.
- Poder crear un sistema de rol propio, teniendo en cuenta las necesidades de este proyecto.
- Adquirir experiencia en el uso de software habitual en el desarrollo de videojuegos y aplicaciones móvil, a la vez de formar un punto de vista personal sobre el proceso de desarrollo de estas mismas y de la industria que les rodea.

1.3. Propósito y objetivos del proyecto

El propósito principal de este proyecto es crear un sistema de rol y desarrollar una aplicación móvil conjunta que permita crear y administrar personajes del mismo sistema de manera que se puedan jugar partidas de rol de forma satisfactoria. La aplicación debe simplificar varios sistemas que añaden tedio en las partidas de rol comunes, de forma que agilizan las propias partidas.

Concretamente, estos son los objetivos del proyecto:

- Aprender a crear interfaces de usuario fáciles de entender y usar.
- Aprender a crear los assets necesarios para el proyecto, como fondos y sonidos, de propia mano.
- Aprender a organizar un proyecto grande de manera que añadir modificaciones a posteriori no sea complicado o tedioso.
- Estudiar las bases de los juegos de rol.
- Crear un sistema de rol simple pero eficaz a partir de los conocimientos adquiridos.
- Documentar una guía del sistema de rol con descripciones de cada elemento para los jugadores.
- Crear un sistema de creación de personajes paso a paso del sistema de rol diseñado.
- Crear un sistema de gestión de personajes con la implementación de las mecánicas diseñadas.

1.4. Distribución de tareas

Debido a que este proyecto se basa en la creación de un sistema de rol y de una aplicación móvil implementando sus mecánicas, la distribución de las tareas a realizar para su desarrollo es la siguiente:

Estética	15%
Narrativa	5%
Mecánicas	50%
Tecnología	30%

El apartado “Estética” simboliza el trabajo realizado en el diseño de la aplicación y sus interfaces de usuario, juntamente con la creación y adaptación de los assets usados.

El apartado “Narrativa” se atribuye al trasfondo del mundo del sistema de rol que ayuda a los jugadores a situar un contexto para sus partidas. Este apartado muestra ese bajo porcentaje debido a que lo desarrollado no incluye una historia propia ni personajes preestablecidos. La aplicación sirve para simplificar y automatizar mecánicas del juego de rol. Uno de los pilares fundamentales de los juegos de rol de mesa es permitir que los jugadores sean quienes deciden cual es la narrativa que desean seguir y cómo son los personajes que quieren encarnar. Es por esto que varios puntos en el apartado del diseño de la aplicación sugeridos en la guía han sido omitidos o modificados, para adaptarse mejor al tipo de desarrollo de este proyecto.

El apartado “Mecánicas” representa el diseño, testeo y balance de todas las mecánicas que conforman el sistema de rol propio de la aplicación, incluyendo el sistema de combate, el sistema de progresión de personajes, el sistema de habilidades, el sistema de valores, el sistema de acciones, el sistema de razas, el sistema de clases y el sistema de equipamiento.

El último apartado, “Tecnología”, representa la implementación en Unity de todas las mecánicas del sistema de rol diseñado, incluyendo el sistema de creación y el sistema de gestión de personajes en la aplicación móvil. Todo el diseño del código y de las clases viene representado en este apartado.

2. Estudio de viabilidad

Para poder asegurar que este proyecto puede llevarse a cabo sin problemas primero se deben hacer algunas comprobaciones. Estas consisten en investigar los recursos necesarios para completar el proyecto, hacer un estudio de mercado actual para conocer la posible competencia, analizar el modelo de negocio planeado y finalmente investigar el público objetivo de la aplicación resultante.

2.1. Viabilidad y recursos

En este apartado se expondrán los recursos y herramientas utilizadas para el desarrollo del proyecto. Junto a los listados de todos los elementos usados para el desarrollo, se mostrarán sus costes aproximados y se estimará un presupuesto ficticio para valorar la viabilidad económica del proyecto.

2.1.1. Recursos de hardware

- PC de sobremesa custom: Unidad principal de trabajo para el desarrollo del proyecto. Conformada principalmente por un procesador AMD Ryzen 9 3950X, 32 GB de ram y una procesador gráfico Nvidia GeForce RTX 3070.
  Coste total: 2.455€.

- Portátil Lenovo Legion Y520: Unidad secundaria de trabajo para el desarrollo del proyecto. Usada en caso de no tener acceso a la unidad principal. Conformada principalmente por un procesador Intel i7-7700HQ, 8 GB de ram y un procesador gráfico Nvidia GeForce GTX 1050.
  Coste total: 899€.

- Smartphone Xiaomi Mi A3: Móvil personal utilizado durante el proyecto como dispositivo de pruebas de la aplicación.
  Coste total: 189€.

  Todos estos recursos se adquirieron anteriormente al proyecto, por lo tanto, no han supuesto un gasto económico para este mismo.

2.1.2. Recursos de software

- Unity 2020.3.24f1: Motor de videojuegos usado para el desarrollo de la aplicación.
  Coste: Gratuito.

- Adobe Photoshop: Programa de edición de imágenes usado para la edición y creación de fondos y otros elementos para la aplicación.
  Coste: 290,17€.

- Inkscape: Programa de dibujo vectorial usado para la confección de iconos para la aplicación.
  Coste: Gratuito.

- Visual Studio 2019: Editor de código principal usado por defecto por Unity. Usado para programar la aplicación y debugar.
  Coste: Gratuito.

- GitHub Desktop: Versión local de GitHub usada para almacenar el proyecto de forma segura.
  Coste: Gratuito.

- Google Drive: Servicio de almacenamiento en la nube usado para guardar el proyecto de forma segura y documentarlo.
  Coste: Gratuito.

- Convertio: Servicio web usado para convertir archivos de un formato a otro.
  Coste: Gratuito.

- 123Apps: Servicio web usado para la edición de audio entre otras herramientas.
  Coste: Gratuito.

Todos estos recursos son gratuitos o se adquirieron anteriormente al proyecto, por lo tanto, no han supuesto un gasto económico para este mismo.

2.1.3. Recursos humanos

Para desarrollar cualquier proyecto se requiere de un grupo de profesionales, a poder ser especializados en las áreas que se

necesiten, para la correcta creación de dicho proyecto. En este caso, se estima que los roles profesionales estrictamente necesarios para este proyecto son:

- Desarrollador de apps Android: Encargado de crear la aplicación móvil. Programar y diseñar dicha aplicación va a cargo suyo.
  Coste: 10,17€/hora.

- Artista gráfico: Encargado de diseñar assets gráficos para la aplicación y el manual del sistema de rol.
  Coste: 30€/hora.

- Diseñador de juegos de rol: Encargado de confeccionar todas las mecánicas necesarias del sistema de rol esencial para el proyecto.
  Coste: 30€/hora.

Valores obtenidos en la web https://www.payscale.com/

Estos no son los únicos roles para poder llevar a cabo el proyecto, pero sí que son los más imprescindibles. En el caso actual, el proyecto se ha realizado por una persona, que ha tenido que encargarse de todos los roles necesarios para completarlo. Aparte, han habido testers ajenos al desarrollo del proyecto para recibir opiniones sobre la aplicación.

2.1.4. Viabilidad económica

En el caso de este proyecto, ya se han explicado en los puntos anteriores que los costes tanto de hardware, como software y recursos humanos son nulos ya que todos los recursos ya habían sido adquiridos previamente.

Si ese no fuera el caso e hipotéticamente sí fuese necesario adquirir todos esos recursos, esta sería una estimación resumida del coste de desarrollo del proyecto:

RECURSO	COSTE
Hardware	3.543€
Software	290,17€
Recursos humanos	64.677€
TOTAL	**68.510,17€**

Los costes de los recursos humanos de la estimación se han calculado a partir del sueldo medio anual en España de cada rol estipulado en el punto anterior.

2.4. Estudio de mercado

Antes de poder empezar el desarrollo del proyecto, primero se deben investigar otros proyectos de similar índole en el mercado actual para conocer los puntos fuertes que replicar y los puntos débiles que aprovechar. También se estudian los modelos de negocio de la competencia para ajustar el posible modelo del proyecto de forma adecuada.

2.4.1. Estado del arte

Es muy importante durante el desarrollo de un proyecto conocer a la posible competencia y aprender los aspectos positivos y negativos de la misma para poder así crear un producto que destaque entre esa misma competencia y así atraer parte de la cuota de mercado. Normalmente suplir alguna carencia importante o solucionar aspectos negativos de la competencia en el propio proyecto suelen ser razones sustanciales para poder distinguirse del resto. En este caso, se han analizado aplicaciones disponibles de forma gratuita en la Google Play Store que ofrecen una forma de crear y gestionar personajes basados en el sistema de rol Dragones y Mazmorras.

Esta es la selección de aplicaciones que se han analizado con profundidad y de las que se ha tomado nota de varios aspectos para desarrollar la aplicación de este proyecto:

D&D Beyond

Aplicación creada por los creadores de las últimas ediciones del sistema Dragones y Mazmorras. Ofrece multitud de funcionalidades, incluidas la creación y gestión de personajes y sirve como versión portátil de su aplicación web del sistema. De esta aplicación se debe destacar el diseño gráfico usado, y su sistema de creación de personajes, que ofrece múltiples formas de crearlos, aunque en este caso el modo de creación paso a paso será lo que este proyecto aprovechará.

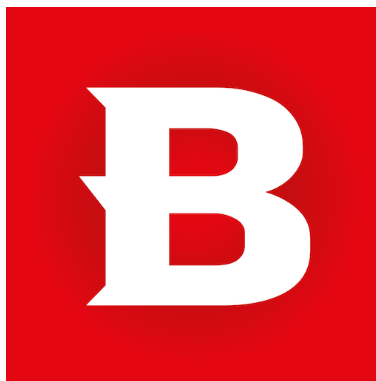

Figura 2.4.1/1: Icono de la aplicación D&D Beyond

Fight Club 5

Aplicación centrada en la quinta edición de Dragones y Mazmorras. Solo ofrece el creador y gestor de personaje. En esta aplicación la información del personaje es mucho más clara, concisa y accesible, siendo muy notable esa decisión de diseño en el proceso de creación de personaje. Como lado negativo, su versión gratuita solo permite tener un personaje. La característica de la información es el detalle a seguir en este proyecto.

Figura 2.4.1/2: Icono de la aplicación Fight Club 5

5e Character Keep

Otra aplicación basada en la quinta edición de Dragones y Mazmorras. Similar al ejemplo anterior, ofrece los sistemas de

creación y gestión de personajes, esta vez hasta 3 personajes, además de la opción de añadir todo tipo de datos personalizados, como razas o hechizos al gusto. En este caso hemos tomado de referencia parte del diseño del creador de personajes.

Figura 2.4.1/3: Icono de la aplicación 5e Character Keep

Character Sheet for any RPG

Aplicación que permite crear personajes basándose en diversos sistemas de rol diferentes. Desgraciadamente, esta aplicación no ofrece un creador de personaje paso por paso, sino que al seleccionar el sistema en el que se basa, pasa a mostrar su ficha, donde el jugador ya puede modificarla libremente. Es la aplicación que menos ayuda al usuario a entenderla, pero a cambio organiza la ficha del personaje de forma clara. Esta es la característica a tomar en cuenta para este proyecto.

Figura 2.4.1/4: Icono de la aplicación Character Sheet for any RPG

2.4.2. Comparación del estado del arte

Para esclarecer lo que ofrece cada aplicación nombrada anteriormente y compararlo con las características pensadas de aplicación a desarrollar, se usa la tabla a continuación.

	Sistema de rol	Creador de personajes	Número de personajes	Versión Premium
D&D Beyond	D&D	Varios	6	Sí
Fight Club 5	D&D	Por Pasos	1	Sí
5e Character Keep	D&D	Por pasos	3	Sí
Character Sheet for any RPG	Múltiples	No	Sin límite	No

Con esta información recopilada, podemos considerar las siguientes decisiones:

- El sistema usado en la aplicación a desarrollar debe ser diferente al resto para reducir la cantidad de opciones y así no abrumar a los jugadores nuevos.
- Es mejor usar un creador de personajes por pasos para que los usuarios entiendan qué están eligiendo y no abrumarlos con las múltiples elecciones.
- Parece razonable permitir crear un número límite de personajes en caso de querer crear una versión premium de pago para generar ingresos. En este caso, 4 es un número racional comparado con la competencia.
- Es importante que la navegabilidad dentro del gestor del personaje sea fácil de entender, mostrando a poder ser todo un tipo de información en una sola pantalla.
- Tener múltiples formas de navegación puede llegar a desorientar al usuario. Hay que mantener una forma de acceder a toda la información.
- En el momento de crear un personaje, si se dan muchas opciones al jugador puede provocar rechazo. Mejor seguir diseño por sustracción.

2.4.3. Modelo de negocio

La monetización en una aplicación móvil Android suele tener muchas alternativas. A continuación se listan las más populares dentro de las apps vinculadas a sistemas de rol:

- Ofrecer funcionalidades extra a cambio de un pago: Se basa en ofrecer al usuario, en este caso al jugador, un número de funcionalidades extra que puedan ser de su

interés a cambio de un pago. La aplicación de base es gratuíta, pero hay funcionalidades que sólo pueden ser usadas tras pagar.

- Acceso gratuíto con publicidad: Se basa en tener publicidad dentro de la aplicación, y ofrecer eliminar esa publicidad a cambio de un pago.
- Aplicación de pago: El acceso a la aplicación solo puede conseguirse a través de un pago.
- Compras en la aplicación: Se basa en ofrecer un número de productos relacionados con el sistema de rol, ya sean manuales o guías a los usuarios para que los compren a través de la app.

Así pues, el sistema de monetización elegido es ofrecer una aplicación gratuíta, que es la que se desarrolla en el proyecto y a parte, otra aplicación de pago con funcionalidades extra.

Al ser una aplicación desarrollada para Android, se publicaría en la Google Play Store de forma gratuita. Al cabo de un tiempo y dependiendo de su popularidad, se publicaría la versión mejorada con funcionalidades extra, esta aplicación siendo de pago. Ambas aplicaciones recibirían actualizaciones periódicas solucionando errores, mejorando funcionalidades existentes o incluso añadiendo de nuevas, por lo tanto, requeriría un mantenimiento constante.

En lo que refiere a la aplicación gratuita, se ha diseñado para que todos los jugadores del sistema siempre tengan acceso a las herramientas que ofrece la aplicación y priorizar una mayor cuota de mercado. Por decisión de diseño y comodidad para los usuarios, se decide que la aplicación no tendrá anuncios dentro.
Por otra parte, también se vendería el manual del sistema de rol en formato físico a cambio de un precio un poco más alto al de sus costes de producción, para así asegurar una mayor accesibilidad para los jugadores. Eventualmente, se venderían también en formato físico o en las mismas aplicaciones guías con campañas, mapas, personajes y enemigos ya confeccionados para que los jugadores puedan jugar sin tener que preocuparse de ese aspecto.

2.5. Público objetivo y perfil de jugador

Para poder llevar a cabo un proyecto como este, es necesario acotar el público objetivo al que dirigir el producto final. En este caso, el objetivo es ofrecer una alternativa de menor complejidad que el resto de sistemas de rol actuales, por ende, el público objetivo puede describirse como todos esos interesados por el mundo del rol, tanto adolescentes como adultos, que buscan una forma de iniciarse en este pasatiempo. Esto tampoco excluye a la gente con experiencia previa en los juegos de rol, pero sí es cierto que el objetivo base del proyecto se dirige especialmente a aquellos que no han probado aún un juego de esa índole.

Se puede especificar el público potencial según la experiencia que buscan como jugadores usando la taxonomía de Bartle, creada en 1996 por Richard Bartle para clasificar a los tipos de jugador de juegos en línea según sus motivaciones. Según esta clasificación, existen:

- Socializadores: Jugadores que su mayor motivación es poder interactuar con el resto de jugadores, a veces incluso interpretando la personalidad del personaje que controlan, sin importarles el objetivo principal del juego.
- Asesinos: Jugadores agresivos que su principal objetivo es ser hostil contra otros jugadores o superarlos. Suelen ser mucho más competitivos que el resto, su mayor meta siendo ser el mejor en ese juego.
- Cumplidores: Jugadores los cuales su mayor propósito es completar todo lo que el juego les ofrezca, obtener las mayores puntuaciones posibles o acumular el mayor número de recursos posible. Suelen interactuar más con el mundo del juego.
- Exploradores: Jugadores que su objetivo principal es poder explorar todos los rincones del mundo del juego, conocer sus lugares y encontrar los secretos mejor escondidos. Su interacción se decanta más por la interacción con el mundo.

Los juegos de rol de mesa dan plena libertad a que participen todo tipo de jugadores, aunque si tenemos en cuenta esta taxonomía, los "socializadores" son el tipo de jugador con mayor probabilidad a jugar juegos de rol. Su estilo de juego se alinea completamente con la base jugable del rol, que es interpretar a un personaje e interactuar con el resto de personajes y jugadores. Tampoco se puede descartar el

arquetipo de “exploradores”, ya que ese anhelo por explorar cada rincón suele ser beneficioso durante las partidas de rol y es integral para poder entender el entorno en el que se encuentran en cada situación.

3. Planificación

En todo proyecto siempre se establece una planificación tomando en cuenta el tiempo disponible y las tareas requeridas para completar dicho proyecto. Es por eso que en este apartado se concretarán todas las tareas a completar y se organizarán delimitando el tiempo estimado de compleción dentro del período establecido. Para poder gestionar y completar todas esas tareas con los marcos de tiempo disponibles de forma eficaz, se establecerá una metodología de trabajo concreta.

3.1. Tareas planificadas

A continuación se listarán todas las tareas principales y sus subtareas que conforman el proyecto.

- **Planificación del proyecto:**
  - Ideación del proyecto.
  - Estudio de mercado.
  - Estudio de viabilidad.

- **Diseño del sistema de rol:**
  - Diseño del sistema de valores.
  - Diseño del sistema de razas.
  - Diseño del sistema de clases.
  - Diseño del sistema de combate.
  - Diseño del sistema de habilidades.
  - Diseño del sistema de acciones.
  - Diseño del sistema de progresión.
  - Diseño del sistema económico.
  - Diseño del trasfondo.
  - Diseño del sistema de inventario.

- **Diseño de la app:**
  - Diseño del menú inicial.
  - Diseño de los menús del creador de personajes.
  - Diseño de los menús del gestor de personajes.
  - Diseño de sonidos.

- **Implementación de la app.**
  - Implementación del sistema de valores.

- Implementación del sistema de razas.
- Implementación del sistema de clases.
- Implementación del sistema de combate.
- Implementación del sistema de habilidades.
- Implementación del sistema de acciones.
- Implementación del sistema de progresión.
- Implementación del sistema económico.
- Implementación del sistema de inventario.
- Implementación del sistema de creación de personajes.
- Implementación del sistema de guardado.
- Implementación del menú inicial.
- Implementación de los menús del creador de personajes.
- Implementación de los menús del gestor de personajes.
- Depuración de errores.

- **Documentación:**
    - Redacción del manual del sistema de rol Animaia.
    - Redacción de la memoria y resumen del proyecto.

3.2. Organización de las tareas

Ahora se mostrará la forma en la que se han organizado dichas tareas.

Semana 1 Agosto 2022 - Semana 1 Octubre 2022:
- Planificación del proyecto.

Semana 3 Agosto 2022 - Semana 2 Enero 2023:
- Diseño del sistema de rol.

Semana 1 Noviembre 2022 - Semana 1 Abril 2023:
- Diseño de la app.

Semana 3 Noviembre 2022 - Semana 3 Abril 2023:
- Implementación de la app.

Semana 1 Septiembre 2023 - Semana 1 Junio 2023:
- Documentación.

Figura 3.2/1: Diagrama de Gantt representando la planificación de las tareas del proyecto.

3.3. Metodología de trabajo

Para poder organizar dichas tareas de forma eficiente, se decide que este proyecto seguirá la metodología de SCRUM.

Esta metodología permite organizarse a medida que se avanza en el proyecto en contra de organizarlo todo antes de empezar, lo que posibilita un desarrollo más flexible y consecuentemente, poder mejorar el producto de manera más fácil.

Esta metodología funciona de la siguiente manera. Primero de todo se planean todos los elementos que se quieren desarrollar en el proyecto y se les dan una prioridad. Entonces, con los elementos de mayor prioridad se idea el producto mínimo viable que servirá como meta mínima de desarrollo. A partir de aquí, se empiezan a desarrollar las partes de ese producto mínimo viable una por una en periodos de unas 2 semanas llamados sprints. Entre sprint y sprint, se plantea cuál será la próxima parte a implementar. A medida que cada parte termina su desarrollo mínimo, entonces se le añaden las mejoras de menor prioridad para aumentar la calidad del producto. Cada parte no tiene porque seguir un orden de desarrollo estricto, sino que dependiendo de la situación, se reorganizan las prioridades y entonces se decide con qué se trabajara durante el siguiente sprint. Periódicamente se van haciendo reuniones para mostrar los avances de cada sprint y poder estructurar y gestionar el resto de trabajo.

En el caso de este proyecto, estos son los pasos a seguir:

1. Primeramente se plantea el sistema de rol en el que se va a basar la aplicación; se diseñan las bases del sistema sin entrar en detalles.
2. Teniendo el esqueleto que permite la funcionalidad básica del sistema de rol, se empieza a trabajar en la aplicación.
3. Se diseñan los menús de forma simple, con la estructura de la interfaz planteada.
4. Teniendo idea de donde se verán los elementos en pantalla, entonces se les añade funcionalidad a esa interfaz.
5. En el momento de tener esos menús funcionales, entonces se le implementan los pertinentes sistemas del sistema de rol.
6. Cuando la aplicación ya sirve con las funcionalidades básicas, entonces se añaden detalles al sistema de rol.
7. Se implementan esos detalles nuevos en la aplicación.
8. Luego de tener eso hecho, entonces se añaden funcionalidades nuevas a la propia aplicación.
9. Al tener en la aplicación el sistema de rol ya implementado completamente y todas las funcionalidades creadas, finalmente se acaba el diseño de la aplicación y se termina su desarrollo para el proyecto.

4. Marco de trabajo y conceptos previos

Antes de explicar el diseño e implementación del proyecto, es necesario desglosar todos los conceptos necesarios para el correcto entendimiento de esta memoria. Aparte, también se enumerarán de forma detallada todos los programas usados durante el desarrollo del propio proyecto.

4.1. Conceptos previos

Este proyecto está basado en un pasatiempo que no es muy popular por el público general. Por lo tanto, hablar de elementos que forman parte del proyecto puede resultar confuso en algunos aspectos para los lectores sin conocimientos previos, debido al lenguaje propio que suele usarse en el rol.

- Tirada de dados: En los sistemas de rol, para determinar el resultado de una acción, ya sea un ataque o intentar convencer a alguien, por ejemplo, se usan dados poliédricos que al tirar muestran un número en la cara superior. Dicho número decide el resultado de la acción ejecutada. Cuando se refiere a un dado en concreto suele llamarse como “d” + número de caras. El dado clásico de 6 caras, por ejemplo, se llamaría d6. Ver **Figura 4.1/1**.

Figura 4.1/1: Muestra de diferentes dados poliédricos.

- Director de partida/Game Master/DP/GM: Hace referencia al jugador que se dedica a preparar la historia de las partidas y a controlar los personajes que el resto de jugadores no controla, como los enemigos o NPCs.

- NPC/PNJ: De las siglas Personaje No Jugable en inglés (Non-Playable Character), se refiere a todos los personajes que sólo el director de partida puede controlar.

- Ficha de personaje: Toda la información que forma parte de un personaje dentro de un juego de rol está en su ficha de personaje. Ver **Figura 4.1/2**. En el caso del proyecto, el gestor de personaje puede considerarse propiamente como una ficha de personaje.

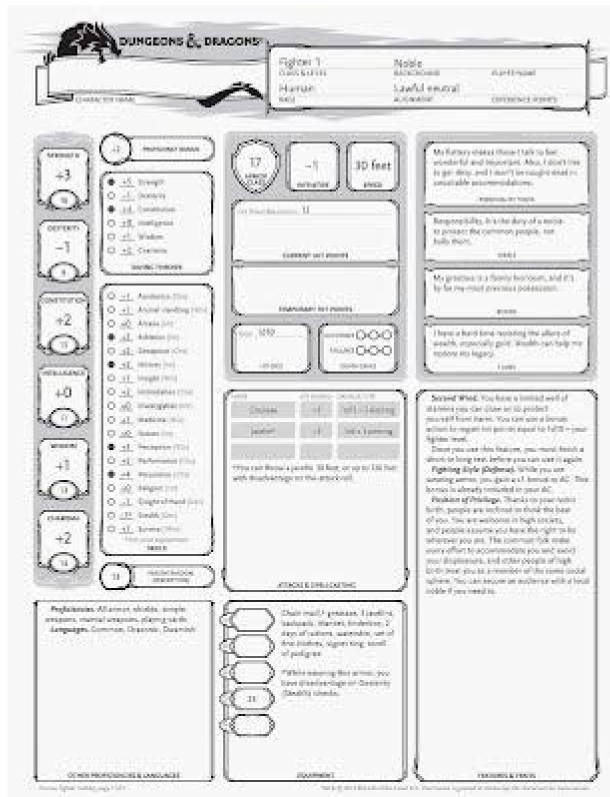

Figura 4.1/2: Ficha de personaje del sistema Dragones y Mazmorras.

- Rolear: Se refiere a la acción de interpretar el personaje en control teniendo en cuenta la personalidad y pensamientos del propio personaje. Es la base de los juegos de rol, actuar con la mente del personaje.

- Bufo: Cuando una habilidad otorga efectos positivos, como aumentar el ataque o la defensa de los aliados, esos efectos se consideran bufos. El orígen de la palabra viene del inglés *buff*, que significa hacer más poderoso algo o alguien.

- Debufo: Contrario al concepto anterior, simbolizan los efectos negativos que afectan a algo o alguien. Su orígen es el mismo de bufo, pero con el prefijo inglés negativo *de-*, *debuff*.

- Prefab: Es un tipo de asset que permite almacenar un elemento con sus componentes y propiedades y sirve como plantilla a partir de la cual crear nuevas instancias del elemento. Cualquier edición hecha a un prefab actualiza el resto de en instancias producidas por él, pero, también permite ajustes para cada instancia individualmente.

- Asset: Se refiere a un elemento que puede ser usado en un proyecto, ya sea una imagen, un modelo 3D, un audio o un objeto cualquiera.

4.2. Entornos de trabajo

Anteriormente, en el punto 2.1.2. Recursos de software, se han nombrado todos los programas usados para el desarrollo del proyecto de forma concisa. En este punto se dará una explicación más detallada de la razón de uso de cada entorno de trabajo.

4.2.1. Entorno de implementación

Para implementar el sistema de rol en una aplicación para móviles se tomaron en consideración muchas opciones, siendo las más idóneas Android Studio y Unity debido a la experiencia previa conseguida en proyectos anteriores del GDDV. De entre las 2, se decidió usar Unity, en concreto la versión 2020.3.24f1. Esa decisión fue tomada teniendo en cuenta la experiencia total y conocimientos previos al proyecto que se tenían del motor, ya que había sido usado en más proyectos que Android Studio y por ende, era una herramienta más conocida para el desarrollo.

Figura 4.2.1/1: A la izquierda, logotipo de Unity. A la derecha, logotipo de Android Studio.

Unity es un motor de videojuegos 2D o 3D multiplataforma creado por la compañía Unity Technologies. Disponible en

Windows, Linux y Mac OS, da soporte de compilación en múltiples plataformas, entre ellas PC, consolas de anterior y actual generación, Android, iOS y WebGL. Además, es gratuito siempre y cuando no se supere el límite anual de ingresos de 100.000 $. En ese caso, el desarrollador debe adquirir la versión de pago del motor.

Aunque Unity esté pensado para el desarrollo de videojuegos, su flexibilidad permite el desarrollo de aplicaciones móviles como la del proyecto. Además, el lenguaje de programación estándar de Unity, C#, es uno de los lenguajes con el que más experiencia se tenía con anterioridad. En cuanto la programación, Unity usa Visual Studio como editor de código por defecto, y es el que se ha usado para el desarrollo de la aplicación. Unity cuenta con una gran comunidad y documentación, lo que hace desarrollar en este entorno una faena más cómoda.

Figura 4.2.1/2: Logotipo de Visual Studio.

4.2.2. Entornos de diseño

La gran mayoría de elementos gráficos para la aplicación han sido creados utilizando software de diseño de diversa índole.

Para crear las imágenes de fondo de la aplicación se ha usado Adobe Photoshop. Este programa es el más usado para la creación y edición de imágenes, y Unity permite la edición de imágenes desde Photoshop mediante el uso del asset PSD Importer, el cual genera y actualiza prefabs de las imágenes al mismo tiempo que se editan en Photoshop. Eso, junto a la gran multitud de herramientas que otorga, son las razones de su uso para el desarrollo del proyecto. Aunque sea de pago, ya se poseía una licencia previamente al inicio del proyecto, por lo tanto, el pago no interfirió en la decisión del uso de Photoshop.

Figura 4.2.2/1: Logotipo de Adobe Photoshop.

Inkscape se ha usado para la creación de iconos para la aplicación. Inkscape es un editor de gráficos vectoriales libre y de código abierto. Es por eso que también es gratuito. Este programa suele usarse para el diseño de diagramas, gráficos y logotipos entre muchos más productos. Aunque su desarrollo está enfocado para GNU/Linux, también está disponible en otras plataformas como Windows o Mac OS. La razón principal de su uso en este proyecto es la accesibilidad del programa, sus herramientas disponibles y por la experiencia previa adquirida usandolo en el GDDV.

Figura 4.2.2/2: Logotipo de Inkscape.

Para el sonido, se ha usado el servicio web 123Apps. Esta web de uso libre ofrece herramientas de vídeo, audio, PDF y convertidores de formato de archivos con un límite de tamaño de forma gratuita. En el caso de este proyecto, los archivos de audio con los que se ha trabajado son ligeros, así que el uso de esta web no aporta inconvenientes.

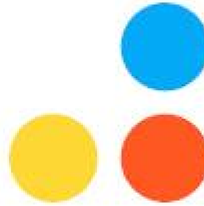

Figura 4.2.2/3: Logotipo de 123Apps.

4.2.3. Entornos de documentación

Para poder documentar la memoria, manual del sistema de rol y apuntar ideas durante el proceso del proyecto, se ha usado Google Docs. Google Docs es un editor de texto en línea que forma parte de la suite Google Docs Editors. Esta aplicación permite el uso de otras aplicaciones de Google conjuntamente en el documento como tablas, gráficos y dibujos. Además, al ser una aplicación web que usa Google Cloud, todos los cambios en los documentos se guardan casi al instante siempre y cuando se tenga conexión a internet.

Figura 4.2.3/1: Logotipo de Google Docs.

4.2.4. Entornos de planificación y gestión

La planificación inicial del proyecto y la realización del diagrama de Gantt han sido realizadas con Gantt Project por ser el programa más idóneo y accesible para su cometido. Este programa de código abierto sirve para la administración de proyectos usando el diagrama de Gantt y está disponible en sistemas operativos como Windows, Linux y Mac OS X. También es gratuito.

Figura 4.2.4/1: Logotipo de Gantt Project.

En cuanto a la gestión del proyecto, se han usado 2 herramientas diferentes. La herramienta principal ha sido Google Drive. Este es un servicio de alojamiento y sincronización de archivos desarrollado por Google que permite a sus usuarios almacenar archivos en la nube, sincronizarlos entre dispositivos y compartirlos. Este servicio se ha usado para almacenar el proyecto y la documentación.

Figura 4.2.4/2: Logotipo de Google Drive.

La secundaria ha sido GitHub, a través de la aplicación GitHub Desktop. Similar en funcionamiento a Google Drive, GitHub es usado para poder almacenar y compartir proyectos de software. GitHub Desktop es una aplicación para sistemas operativos que permite hacer lo mismo pero usando una interfaz de usuario en vez de comandos de consola. En el caso actual, se ha usado para almacenar el proyecto de Unity de la aplicación.

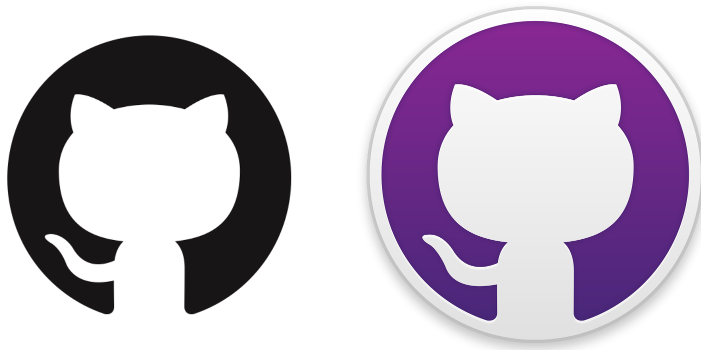

Figura 4.2.4/3: A la izquierda, logotipo de Github. A la derecha, logotipo de Github Desktop.

5. Diseño del proyecto

En este apartado se explicarán las decisiones de diseño tanto del sistema de rol Animaia como de la aplicación móvil. Todos los elementos que conforman ambas partes del proyecto serán comentadas, dando las razones de su creación y utilidades.

5.1. Glosario del diseño

El sistema de rol desarrollado en este proyecto posee varios términos propios los cuales son necesarios de describir antes de comentar el diseño de sus elementos. He aquí un listado de los términos más importantes a conocer:

- Animaia: Nombre que comparte el sistema de rol con el mundo en el que se basa. El origen de este nombre viene de la fusión de los nombres *Ánima*, alma en latín y *Gaia*, nombre de la titánide madre del planeta Tierra en la mitología griega y planeta en el que se basa el sistema de rol "Anima: Beyond Fantasy", sistema del que este propio sistema toma varias referencias. El nombre Animaia también sirve como un juego de palabras que significan "Planeta de las almas".

- Almático/almática: Referente al alma en el sistema de rol.

- Número 4: El diseño del sistema de rol sigue un patrón del uso del número 4 o múltiplos de éste en gran parte de sus elementos como una elección arbitraria de diseño.

- Poder almático/Energía almática/Alma: Son la base de los poderes sobrenaturales que poseen los personajes del sistema de rol. También representa el valor "Poder" de los personajes.

5.2. Mecánicas

Con varios términos intrínsecos del proyecto explicados, se da paso a la explicación de la base del sistema de rol, las mecánicas que lo conforman así como la forma de jugarlo. En el presente punto se comentarán todos los elementos que configuran el sistema de rol

Animaia, desde los personajes que pueden controlar los jugadores hasta el sistema de combate.

5.2.1. Flujo de juego

Las partidas de Animaia siguen un patrón muy característico que suele repetirse continuamente durante una partida, y que describe la forma de jugar.

Primero el GM debe describir el entorno que rodea a los jugadores con el máximo detalle posible. Explicar qué es lo que hay alrededor del grupo como objetos destacables, estructura, materiales, gente del lugar, el tipo de ambiente o puertas por donde seguir explorando, ayuda a los jugadores a imaginar el sitio en el que están y les avivará la curiosidad para interactuar con diferentes partes del entorno.

Luego, los jugadores deciden qué acciones tomar. Pueden hacer lo que les plazca. Cuando decidan hacer algo, se lo comunicarán al GM con sus intenciones y describiendo como quieren hacer lo que quieran hacer. Pueden explicar qué harán de forma individual o pueden hacer algo de forma grupal. Para ejecutar las acciones que quieran hacer pueden seguir un orden establecido o no, pero siempre deben organizarse para describir sus intenciones al GM para que este decida qué ocurrirá. Habrán acciones sencillas que los jugadores podrán hacer simplemente declarándolo, sin la necesidad de probar nada pero habrá otras en las que dicha acción conlleve cierta dificultad que impondrá el GM si lo ve necesario.

Finalmente y justo después de que un jugador haya descrito la acción que quiera hacer, el GM simplemente describirá el resultado de la acción si es simple, pero en caso contrario, el GM decidirá la dificultad que conllevará hacer dicha acción de forma satisfactoria y que deberá superar el jugador con una tirada de dados. Después de la tirada, el GM decidirá el resultado de la acción dependiendo del resultado conseguido

en dicha tirada de dados y narrará las consecuencias del resultado.

Este patrón se perpetúa en la gran mayoría del tiempo durante las partidas, siendo más notable durante un combate. En el combate, los jugadores deberán decidir qué hacer de una forma estrictamente ordenada debido a que usa un sistema de turnos. Fuera del combate, el orden puede ser más flexible, fluido y adaptable para las circunstancias de la aventura. En la **Figura 5.2.1/1** podemos ver ese patrón básico en forma de diagrama.

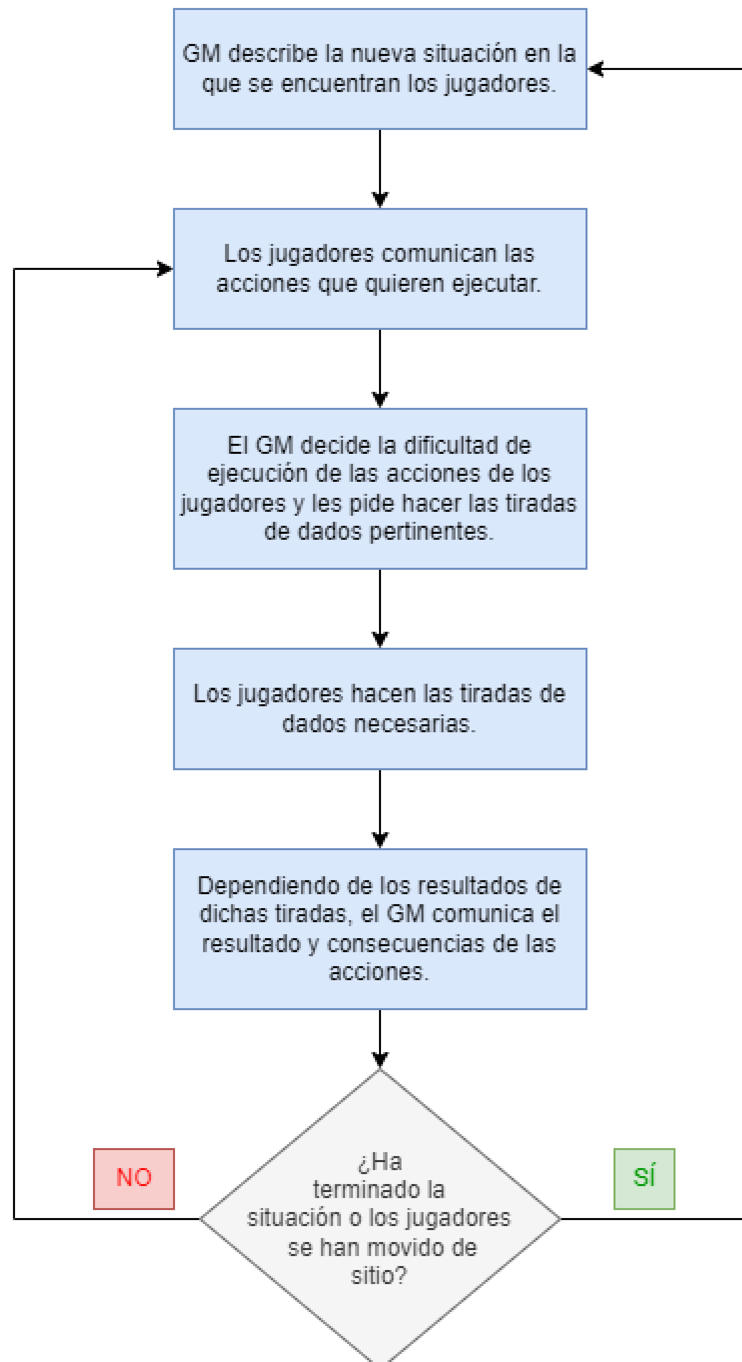

Figura 5.2.1/1: Diagrama del flujo de juego básico.

5.2.2. Personajes

Los personajes en un sistema de rol son el pilar principal de la jugabilidad. El objetivo principal de este tipo de juegos no es más que interpretar un personaje que les sirva a los jugadores como avatar dentro del mundo del juego y crear una historia increíble, proyectando a través de esos personajes sus intenciones durante las partidas.

Para lograr ser ese avatar por el cual los jugadores interactúan con el mundo, los personajes deben tener varias propiedades que lo permitan, ofreciendo herramientas de diferente tipo para que los jugadores puedan actuar de una forma lógica dentro del mundo de juego. Esas propiedades son los valores, la raza, la clase, el equipo y las habilidades. Todas estas propiedades inherentes de los personajes se comentarán más adelante de forma individual.

5.2.2.1. Creación de personajes

Para empezar a crear un personaje, el jugador primero tiene que conocer y decidir cuáles serán sus valores básicos, sus aptitudes mínimas, tanto en cuerpo como en alma, para darle forma desde cero. Los valores básicos serán las estadísticas en las que se basará su nuevo personaje.

Para vincular los valores del personaje, el jugador tendrá que tirar 8 veces, una vez por cada tipo de valor existente en el sistema, 4 dados d4. La suma de cada tirada de los 4 dados d4 será uno de los valores del personaje. Una vez se hayan hecho las 8 tiradas, el jugador podrá distribuir los resultados de dichas tiradas entre los diferentes valores según su criterio.

En caso de haber tenido una tirada con un resultado menor a 6, se puede volver a tirar para conseguir un valor igual o superior a 6. En caso de tener 3 o más tiradas con resultados menores de 8, se pueden hacer otras tiradas para sustituir y acabar teniendo hasta un máximo de 2 tiradas con resultados menores de 8. Esto es para evitar que el nuevo personaje sea incompetente en demasiados ámbitos y así evitar frustración por parte del jugador.

Que el primer paso sea vincular los valores básicos al personaje sirve como primera toma de contacto del

jugador con los valores, las estadísticas que gobiernan el sistema y que influencian los resultados de todas las acciones que pueden hacer. De esta forma los jugadores pueden generar la plantilla estadística de su personaje según la idea general que tengan de su creación y sus preferencias. Igualmente, si en pasos posteriores, las opciones ofrecidas no encajan completamente con la idea de personaje que el jugador quería, se permite un reinicio de la asignación de los valores originales para poder optimizar estadísticamente el propio personaje.

A partir de aquí se empieza a especializar el personaje, primero eligiendo la raza a la que pertenece. La raza aporta valores extra al personaje, que simbolizan las aptitudes inherentes de dicha raza y además le añade un trasfondo que puede afectar con las interacciones entre el resto de personajes debido a las relaciones existentes entre razas del mundo del juego.

Luego de elegir la raza, se selecciona la clase. La clase del personaje simboliza su estilo de juego y por lo tanto, el rol que aporta al equipo. Existe una clase por cada valor existente, y al escoger una clase determinada, su valor distintivo aumenta 2 puntos. Cada clase determina no solo el valor principal del personaje, sino también el listado de armas que puede usar y las habilidades que puede aprender.

Entonces, el siguiente paso es elegir el arma que acompañará al personaje durante la campaña. Esa arma podrá tener nombre propio para que haya un vínculo más personal entre arma y personaje y será la única que podrá usar en toda la aventura. Las armas a elegir estarán determinadas por la clase seleccionada del personaje.

El siguiente paso es elegir las “*acciones ventaja*”, acciones en las que el jugador quiere que el personaje sea proficiente. Deberá elegir 4 acciones de entre todas las existentes en el sistema de Animaia. A esas acciones, cuando las ejecute, el personaje tendrá una pequeña ventaja a la hora de hacer su tirada de dado. Este paso posibilita una mayor personalización en cuanto a las aptitudes del personaje.

Finalmente, el jugador podrá nombrar a su personaje, terminando así su creación.

Todos estos pasos le permiten al jugador obtener un personaje completamente funcional en el sistema de Animaia, aunque aún falten varios elementos a definir. Estos son:

- Las habilidades que el personaje puede usar, determinadas por la clase del personaje. El número total dependerá del número de huecos de memoria disponibles, que estos a la vez dependen del valor de Memoria del personaje.
- El inventario del personaje, los objetos que lleve encima. Pueden ser objetos que existan en el listado del sistema Animaia o otros objetos no especificados.
- Las monedas que lleve, si el GM lo permite, se puede manipular la cantidad de dinero que lleve encima el personaje en el momento de iniciar la aventura según su trasfondo. Al crearse un personaje nuevo, este siempre llevará 1 moneda de platino de forma predeterminada.
- Las características del personaje. Información sobre diversos aspectos del personaje que determinan su comportamiento, y que tener apuntados ayuda a recordar como hacer una interpretación más precisa del personaje.

Hay que tener en cuenta que un personaje siempre empieza con el nivel 1, y a partir de las experiencias que vive, va subiendo de nivel, mejorando sus valores cada vez más y fortaleciéndose.

Todos estos pasos pueden ordenarse en el diagrama de la **Figura 5.2.2.1/1**.

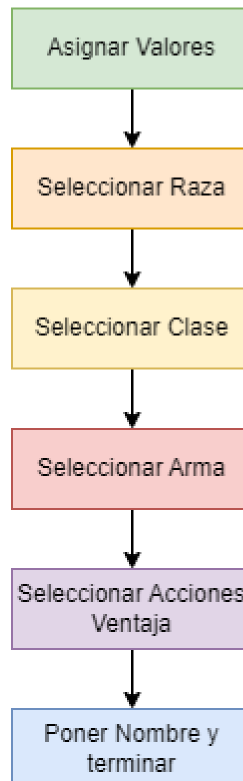

Figura 5.2.2.1/1: Diagrama de los pasos de creación de personaje.

5.2.3. Valores

Todos los personajes tienen asignados unos valores determinados. Hay 8 en total, y representan cada aptitud general del personaje. Estos a la vez se pueden clasificar en 2 grupos, los valores físicos y los valores almáticos.

Los valores del personaje sirven para interpretar las proficiencias de ese personaje en diferentes aspectos. Estos definen como es el personaje, sus cualidades físicas y mentales. Estos valores ayudan a interpretar cómo actúa el personaje ante cualquier situación. Eso es posible gracias a que cada acción que haga el personaje, esté especificada en el sistema o no,

está vinculada a un valor en específico, desde atacar a alguien hasta intentar saltar alto. Todos los valores son representados por un valor numérico que sirve como estadística principal. Ese valor numérico puede ir desde el 4 hasta el 40. Los valores del personaje también están vinculados a los recursos del propio personaje.

Como referencia, el valor base en cualquier valor es 8. Tener un valor menor a ese simboliza una carencia en ese aspecto, mientras que tener un valor superior a 8 simboliza ser mejor que la media en esa faceta.

5.2.3.1. Bonos de valor

Todos los Valores tienen un Bono de Valor asignado. Este depende del número de cada Valor. Los valores superiores a 8 que sean pares añaden +1 a su Bono (10 = +1, 12 = +2, 14 = +3...). Estos bonos sirven para modificar el número conseguido en cada tirada del personaje. Sumando o restando bonos a las tiradas de dados que haga el personaje permite adecuar su habilidad a las acciones ejecutadas. Es la forma que tiene el sistema para representar las proficiencias de los personajes en cada tipo de acción.

Valor numérico	Bono del valor	Valor numérico	Bono del valor	Valor numérico	Bono del valor
4	-2	17	+4	30	+11
5	-2	18	+5	31	+11
6	-1	19	+5	32	+12
7	-1	20	+6	33	+12
8	0	21	+6	34	+13
9	0	22	+7	35	+13
10	+1	23	+7	36	+14
11	+1	24	+8	37	+14
12	+2	25	+8	38	+15

Valor numérico	Bono del valor	Valor numérico	Bono del valor	Valor numérico	Bono del valor
13	+2	26	+9	39	+15
14	+3	27	+9	40	+16
15	+3	28	+10		
16	+4	29	+10		

Figura 5.2.3.1/1: Tabla de equivalencias entre valores numéricos y bonos del valor.

5.2.3.2. Valores físicos

Estos representan las cualidades físicas del personaje. Estas son:

- **Fuerza:** Simboliza cómo de fuerte es un personaje.
- **Resistencia:** Simboliza la vitalidad y resistencia física de un personaje.
- **Agilidad:** Simboliza la facilidad que tiene un personaje para esquivar, hacer acrobacias o movimientos complicados en general. También simboliza la destreza con las manos para hacer cualquier acción.
- **Velocidad:** Simboliza la rapidez que tiene un personaje para moverse o hacer otras acciones.
- **Cuerpo:** Simboliza los puntos de vida del personaje (PC, Puntos de Cuerpo), su vitalidad. El valor del Cuerpo está vinculado a la Resistencia del personaje. El valor base es de 20, y a este se le suma el doble del bono de Resistencia que tenga el personaje.

Los valores físicos sirven para representar las capacidades físicas básicas de una forma más directa. Muchos sistemas de rol intentan representar las capacidades físicas de los personajes a través de estadísticas arbitrarias que representan la capacidad de

golpear fuerte, la capacidad de aguantar golpes y la destreza; pero en este caso, asociar las estadísticas físicas con las capacidades reales es más simple de entender. La única excepción puede ser la agilidad, que representa la flexibilidad en la clasificación de capacidades real, aunque es común asociar la agilidad de alguien con su flexibilidad.

En cuanto al "Cuerpo", este es uno de los recursos principales de los personajes. Este sirve para representar el estado físico del personaje, si sus PC son bajos, su condición física es baja. Esto está hecho así para poder asociarlo fácilmente con el estado de salud del personaje. La decisión del valor base viene del estudio del daño que pueden hacer los personajes a nivel 1. 20 es una cantidad balanceada de vida respecto al daño que pueden llegar a hacer los ataques. Permite aguantar unos cuantos ataques normales de armas regulares a la vez que permite a los jugadores reducirlos a 0 con un muy buen ataque. En cuanto al escalado del Cuerpo respecto a la Resistencia, eso sirve para permitir a ciertos personajes especializarse en soportar golpes en lugar de sus compañeros durante el combate gracias a poder permitirse perder vida sin estar en peligro, lo que es conocido en los juegos de rol como los "Tanques". También permite a todos los personajes poder aumentar sus vidas de forma exponencial si invierten puntos al subir de nivel.

5.2.3.3. Valores almáticos

Representan cualidades de los personajes relacionadas con sus mentes, más específicamente, sus almas. Estas son:

- **Poder:** Simboliza la voluntad del alma que tiene un personaje, la facilidad para ejecutar habilidades, su densidad y su potencia así como el vínculo de sí mismo con el planeta.

- **Sentidos:** Simboliza la perspicacia, intuición y sentidos físicos del personaje.
- **Memoria:** Simboliza la capacidad de obtener conocimiento o de recordar del personaje.
- **Personalidad:** Simboliza el carisma del personaje y su labia.
- **Alma:** Simboliza los puntos de alma del personaje (PA). Para poder usar habilidades, se necesita usar parte de su alma a cambio. El valor Alma está vinculado al Poder del personaje. El valor base es de 30, y a este se le suma el doble del bono de Poder que tenga el personaje.

Los valores almáticos sirven para representar capacidades más abstractas de los personajes como pueden ser la capacidad de ejecutar poderes mágicos, la perspicacia o como de agudos son sus sentidos, el conocimiento que albergan o su carisma.

En cuanto al "Poder", este es otro de los recursos principales de los personajes. Este sirve para representar el estado mental del personaje, si sus PA son bajos, su condición mental es baja. Esto está hecho así para poder asociarlo fácilmente con el estado de salud del personaje. Además, es el recurso utilizado como intercambio para poder ejecutar habilidades. La decisión del valor base viene del estudio del coste de las habilidades existentes. 30 PA permite ejecutar habilidades simples múltiples veces y habilidades fuertes solo una vez antes de que se agoten, pero no posibilita el uso de las habilidades más poderosas sin dejar al usuario sin PA. En cuanto al escalado del Alma respecto al Poder, eso sirve para permitir a ciertos personajes especializarse en el uso de habilidades en combate y a aumentar la cantidad de sus usos si invierten puntos al subir de nivel.

5.2.3.4. Dados de ataque

Representa el número de dados que usa el personaje en el momento de hacer una tirada de dados para atacar con su arma o hacer una habilidad. O sea, si por ejemplo un personaje decide atacar con su arma, al hacer la tirada de ataque, usará el número especificado por este valor de dados. Si su arma usa un d6 y el valor numérico de esta estadística es de 2, el daño del ataque se calculará a partir de la tirada de 2 d6 más el bono del valor vinculado al arma. Esta estadística sirve para que el daño potencial de los personajes aumente sin tener que depender de los valores vinculados a sus ataques. Estos aumentan en 1 dado más sólo cuando el personaje alcanza un nivel par (2, 4, 6, 8, ...).

5.2.3.5. Acciones por turno

Simboliza el número de acciones que un personaje puede hacer por turno. El valor base es de 2, y a este se le suma la mitad menor del bono de Velocidad que tenga el personaje. Este valor permite representar de forma fiel la velocidad de un personaje en el momento de actuar. Cuanto más alto sea el valor de Velocidad, más acciones podrá ejecutar el personaje por turno.

5.2.4. Razas

Como una forma de personalización y clasificación de los personajes, existen las razas. Estas aportan rasgos físicos y de personalidad distintivos para dar variedad a los estilos de personaje además de cierto trasfondo que aporta profundidad al carácter de cada una.

Mecánicamente, las razas otorgan ciertos aumentos en ciertos valores pertenecientes a cada raza. Cada raza tiene un valor principal, uno secundario y otro terciario. Estos valores son los que, cuando un personaje sube de nivel, están disponibles a elegir para aumentar en un punto uno de ellos. El valor principal de la raza aumenta 2 puntos en el personaje cuando se selecciona su raza y el secundario aumenta en un punto. El valor terciario se elige en el momento de seleccionar la raza,

donde se ofrecen 2 opciones. Cada opción de valores de cada raza se verá más adelante. Esto sirve para brindar al jugador una mayor flexibilidad en el momento de crear su personaje. Los valores de cada raza simbolizan sus fortalezas inherentes, inspiradas en las capacidades de los animales en que se inspiran estas razas. A continuación se nombran los valores asociados a cada raza:

- **Reptilianos:** Esta raza está inspirada mayoritariamente en los grandes reptiles como los cocodrilos, dragones de komodo o tortugas.
    - Principal: Resistencia. Asociado a las duras pieles y longevidad de estos reptiles.
    - Secundario: Fuerza. Asociada a la fuerza de las mordidas de cocodrilos o tortugas caimán.
    - Terciario: Agilidad o Personalidad. Asociados a las características de reptiles más pequeños como serpientes o camaleones, respectivamente.

- **Felinos:** Esta raza está inspirada en los animales pertenecientes a la familia de mamíferos felidae, como son los leones, linces, guepardos, tigres o gatos.
    - Principal: Agilidad. Asociada a la gran agilidad y sigilo de estos animales en general.
    - Secundario: Personalidad. Asociada a la creencia popular del ego de los leones o de las marcadas personalidades de algunos gatos.
    - Terciario: Fuerza o Velocidad. Asociadas a la conocida fuerza de los tigres o la extrema velocidad de los guepardos respectivamente.

- **Averios:** Esta raza está inspirada en las aves, principalmente águilas, cuervos, loros y búhos.
    - Principal: Sentidos. Simbolizando los grandes sentidos de muchos de estos animales, como la visión de las águilas o el oído de los búhos.
    - Secundario: Velocidad. Asociada al hecho de pasar largos tiempos volando y las altas velocidades que soportan durante los vuelos.
    - Terciario: Memoria o Poder. Referentes a la conocida memoria perteneciente a los cuervos o el

misticismo que rodea a las lechuzas respectivamente.

- **Muses:** Esta raza está inspirada en los roedores más conocidos como ratones, conejos, castores o ardillas. En este caso, los valores principal y secundario asociados son los restantes entre el resto de razas, pero aún siguen cierta lógica.
    - Principal: Poder. Asociado al uso de patas de conejo como amuletos, de conejos en trucos de magia y en referencia a Mickey Mouse en la película de Disney “Fantasía”.
    - Secundario: Memoria. Asociado a la memoria topológica de las ardillas, una de las más aventajadas del reino animal.
    - Terciario: Resistencia o Sentidos. En referencia a la estructura corporal de los capibaras o los sentidos agudos de las ratas respectivamente.

La raza de los humanos es una excepción. Para ofrecer algo interesante en cuanto a diseño e interpretación, permiten seleccionar los 3 valores asociados de forma libre. Es decir, si se selecciona la raza de humano para su personaje, el jugador podrá seleccionar en orden el valor principal, secundario y terciario libremente. Esto es justificado en el sistema de Animaia para representar el infinito potencial que posee la humanidad.

Otro rasgo distintivo de cada raza son los idiomas inherentes de cada raza, un elemento que sólo aporta más trasfondo a cada personaje.

5.2.5. Clases

La forma de especializar a los personajes en Animaia viene dada a través del sistema de clases. Cada clase se dedica a otorgar un estilo de juego único para los personajes. Esto no limita las acciones que puede hacer cada personaje, pero sí que asigna un listado de tipos de arma que pueden usar o las habilidades que pueden aprender. La forma de especialización en este sistema, similar al resto de sistemas de rol, es a través de las estadísticas del personaje, en este caso, de los valores.

Así pues, cada clase, para poder generar esas especializaciones, se basan en enfocar el aumento de valores al subir de nivel en un valor en concreto. De esta forma se consigue establecer el valor el cual un personaje puede ser aventajado por encima del resto. Cuando un personaje destaca en un valor en concreto, tiende a ejecutar acciones vinculadas a ese valor, por ende, se especializa en ese tipo de acciones.

Cada clase, al ser seleccionada en el proceso de creación de personaje, otorga 2 puntos más al valor al que está asociada, y cuando el personaje sube de nivel, ese valor aumenta automáticamente 2 puntos más, siendo así el valor más fácil de aumentar en cada subida de nivel, solo igualado por algún otro valor que sea perteneciente a la raza del personaje y que se seleccione otra vez como valor adicional. Además, cada clase permite el uso de 6 tipos de armas y el aprendizaje de hasta 8 habilidades como máximo. La restricción de armas y habilidades sirve para no abrumar a los jugadores con muchas opciones a elegir en el momento de crear al personaje, cosa que puede ocurrir en otros sistemas, y para asegurar que el estilo de juego del personaje sigue cierto sentido con sus fortalezas.

Existe una clase por cada valor existente. Estas son:

- **Aniquilador**: Clase especializada en el valor Fuerza. En esta clase los personajes se especializan en el combate para infligir gran daño a sus oponentes. Las habilidades disponibles en esta clase son mayoritariamente ofensivas o se basan en bufos ofensivos y debufos defensivos, y las armas suelen ser armas contundentes o pesadas para poder ejecutar los ataques más poderosos posibles.
- **Protector:** Clase especializada en el valor Resistencia. En esta clase los personajes se basan más en tareas defensivas en combate e incluso curativas, aunque también tienen sus herramientas ofensivas. Las habilidades disponibles en esta clase son mayoritariamente para proteger, bufos defensivos y ofensivos y habilidades para alejar los ataques enemigos del resto de compañeros. Las armas que pueden usar son mayoritariamente armas contundentes o que sirven para poder desviar o bloquear ataques enemigos.

- **Cazador:** Clase basada en el valor Agilidad. Los personajes de esta clase se especializan en el sigilo o en la esquiva y el contraataque durante el combate. Las habilidades disponibles de esta clase se basan en ataques rápidos y repentinos o en bufos y debufos que ayudan a alcanzar a los enemigos más fácilmente. En cuanto a las armas, se basan en armas de cortes precisos o armas punzantes, que suelen requerir un alto nivel de maestría para empuñarlas.
- **Sombra:** Clase basada en la velocidad. Esta clase se fundamenta en los ataques rápidos que no permitan respuesta del rival. Las habilidades disponibles se basan en ataques veloces, combinaciones de ataques o bufos y debufos para dejar a los enemigos sin escapatoria. Las armas que se pueden usar se basan en armas que permiten ataques rápidos.
- **Animatec:** Clase basada en el valor Poder. En esta clase los personajes se especializan en el combate a distancia mediante el uso de habilidades, aunque no se limite a solo eso. Pueden aprender todo tipo de habilidades ofensivas y defensivas, pero ningún bufo o debufo. Las armas que pueden usar son de diversa índole para abarcar el mayor número de posibilidades dentro de las restricciones impuestas en el sistema.
- **Centinela:** Clase basada en el valor Sentidos. En esta clase los personajes no se centran en el combate sinó que en la recopilación de información a través de las acciones vinculadas a sus sentidos. Las habilidades disponibles en esta clase se basan en herramientas para evitar el enfrentamiento directo y el uso de bufos para los compañeros y debufos para los enemigos. En cuanto a las armas, estas se basan en un combate de corto alcance o en el combate a media o larga distancia según las preferencias del personaje.
- **Sabio:** Clase especializada en el valor Memoria. Los personajes pertenecientes a esta clase se centran más en el uso de sus conocimientos para interactuar con el mundo, pero dentro del combate, son una clase bastante polivalente. Las habilidades que aprenden son tanto ofensivas como defensivas, a parte que aprenden bufos y debufos grupales. Las armas que pueden usar se centran

en armas típicamente usadas por combatientes experimentados.
- **Embaucador:** Clase especializada en el valor Personalidad. Esta clase se fundamenta en el uso de la palabra como arma principal, en intentar resolver todo conflicto simplemente conversando y evitar así enfrentamiento directo. Las habilidades disponibles en esta clase se basan solo en bufos y debufos individuales, centrados a un único objetivo. Las armas disponibles se basan en armas sutiles y fáciles de esconder, armas que requieren destreza o armas intimidantes.

Así pues, las clases determinan hasta cierto punto los roles que toman los personajes en sus grupos.

5.2.6. Equipo

Todo personaje dispone de su equipo para la campaña. Su equipo está conformado por su arma predilecta, su inventario donde guarda sus preciadas pertenencias y el monedero, donde guarda su dinero de forma organizada. Esta mecánica permite a los jugadores no solo tener un espacio donde consultar el arma que usa su personaje, sino también un elemento donde guardar objetos que puedan encontrar por ruinas por ejemplo, o llevar consigo objetos necesarios para la interpretación correcta de sus personajes. A parte está el monedero, donde los jugadores pueden consultar y organizar sus riquezas acumuladas. A continuación se describirán todos los elementos que conforman el equipo de un personaje.

5.2.6.1. Armas

Las armas son las herramientas que usan los personajes para combatir. Estas se pueden clasificar de 2 formas, por tipo de arma o por su estilo de uso.

Los 2 tipos son Fuerza y Agilidad, y simbolizan con cuál valor del personaje escala el arma. Cuando un personaje ataca con un arma del tipo Fuerza, el daño que hará el ataque se mejora con el bono del valor Fuerza. Lo mismo ocurre con las armas de Agilidad, en las que sus ataques se mejorarán por el bono del valor de Agilidad del

personaje. Esto permite vincular sus habilidades al uso de sus armas y calcular el daño que infligen con ellas acorde a esas habilidades.

Los estilos de uso son cuerpo a cuerpo o a distancia. La gran mayoría de armas son cuerpo a cuerpo, donde la excepción son la Lanza y la Daga por poder ser lanzadas al enemigo sin la necesidad del personaje de estar enfrentado con ese enemigo, y el Arco, con el que siempre se puede atacar independientemente de la posición. Por como funciona el sistema de combate, la gran mayoría del tiempo se necesita tener a los personajes enfrentados cuerpo a cuerpo con los enemigos para que las peleas sean interesantes y mantengan a los jugadores en el estado de *flow* por tener que arriesgarse y acercarse a los enemigos para atacar, por eso la gran mayoría de armas son de este estilo, pero se dan las excepciones estipuladas anteriormente para dar un poco de variedad a los estilos de combate.

Todo arma, además del tipo, tiene un dado de ataque vinculado con el que se calcula el daño base del ataque. Esto sirve para que las armas infligan un daño mínimo por sí solas, sin tener que depender de los valores del personaje. Cada arma tiene un tipo de dado diferente según el daño potencial que puede infligir en un solo ataque. A continuación se lista todas las armas disponibles:

- **Cuerpo:** El uso de puñetazos, codazos y patadas para atacar.
    - Tipo: Fuerza.
    - Dado de ataque: d6.
- **Espada recta:** Arma de doble filo equilibrada y versátil, útil para cortar o atravesar.
    - Tipo: Fuerza.

  - Dado de ataque: d8.
- **Espada curva:** Arma de un solo y afilado filo curvo para asestar precisos y profundos cortes.
  - Tipo: Agilidad.
  - Dado de ataque: d8.
- **Estoque:** Arma larga y ligera de hoja dura y puntiaguda hecha para perforar pero capaz también de cortar.
  - Tipo: Agilidad.
  - Dado de ataque: d8.
- **Espadón:** Gran espada pesada hecha para obliterar con fuerza bruta a los enemigos.
  - Tipo: Fuerza.
  - Dado de ataque: d12.
- **Hacha:** Arma versátil con una hoja gruesa y pesada unida a un mango capaz de realizar poderosos ataques gracias a su pesada hoja.
  - Tipo: Fuerza.
  - Dado de ataque: d8.
- **Gran Hacha:** Similar a una hacha pero más pesada y con un mango más largo para ataques de barrido devastadores.
  - Tipo: Fuerza.
  - Dado de ataque: d12.
- **Lanza:** Un arma perforante de largo alcance compuesta por una hoja unida a un mango largo. Puede ser lanzada como ataque.
  - Tipo: Agilidad.
  - Dado de ataque: d8.
- **Alabarda:** Arma de asta de mango largo que combina las funciones de hacha y lanza.
  - Tipo: Fuerza.
  - Dado de ataque: d8.
- **Bastón:** Arma sencilla de ataques contundentes muy versátil que consiste en un palo largo.
  - Tipo: Agilidad.
  - Dado de ataque: d8.

- **Arco:** Arma para atacar a distancia disparando flechas.
    - Tipo: Agilidad.
    - Dado de ataque: d6.
- **Daga:** Arma ligera de corto alcance para cortes y puñaladas. Puede ser lanzada como ataque.
    - Tipo: Agilidad.
    - Dado de ataque: d6.

Esta selección de armas en concreto es para permitir a los jugadores elegir entre las armas con más popularidad entre los juegos de rol, añadiendo alguna que otra opción más exótica para balancear los tipos y distanciarse un poco de lo ordinario.

Cada clase tiene acceso a 6 tipos de arma diferentes porque eso permite una repartición equitativa de los tipos de arma entre todas las clases y para ayudar al jugador a centrarse en una elección más específica.

5.2.6.2. Objetos

Son elementos que los personajes llevan en sus aventuras. Los personajes pueden llenar su inventario con objetos ya sean personales, encontrados en los sitios o comprados en tiendas específicas de las poblaciones. También pueden ser vendidos por un precio normalmente inferior al que tienen de compra los mercaderes.

Todos los personajes pueden llevar hasta 8 objetos diferentes. Estos objetos pueden ser objetos de uso común como comida, bebida o utensilios o pueden ser objetos personales importantes para la historia de ese personaje, o objetos importantes de la campaña. Hay libertad para poner los objetos que se quieran siempre y cuando sea lógico dentro de la campaña.

A continuación se dará la lista básica de los objetos disponibles en posibles campañas:

Comida (1 ración)	8 plata	Tiza (50 usos)	2 plata
Bebida (1 litro)	2 plata	Pizarra pequeña	30 plata
Plato	4 plata	Cera para sellar (20 usos)	12 plata
Cubiertos	5 plata	Vela (8 usos)	8 plata
Botella	4 plata	Ropa sencilla	50 plata
Vaso	3 plata	Ropa de viaje	250 plata
Cuerda (10 metros)	10 plata	Ropa elegante	1000 plata
Cadena (10 metros)	25 plata	Disfraz	200 plata
Candado	30 plata	Maquillaje (40 usos)	80 plata
Cofre	100 plata	Peluca:	50 plata
Mochila	20 plata	Perfume (40 usos)	70 plata
Saco	5 plata	Espejo	40 plata
Barril	35 plata	Hilo (50 usos)	12 plata
Cubo	5 plata	Aguja	6 plata
Libro	30 plata	Jabón (24 usos)	20 plata
Papel (10 hojas)	2 plata	Antorcha (6 usos)	25 usos
Sobre	2 plata	Lámpara	60 plata
Tinta (1 uso)	10 plata	Lupa	45 plata
Pluma	20 plata.	Prismáticos	300 plata
Lápiz (100 usos)	4 plata.	Pico	55 plata
Navaja	50 plata	Pala	45 plata
Caña de pescar	150 plata	Amuleto	50 plata
Gancho	100 plata	Pito	10 plata
Red	45 plata	Campanilla	10 plata
Reloj de bolsillo	200 plata	Piedra de afilar	5 plata
Kit médico	65 plata	Muñeca	25 plata

Figura 5.2.6.2.1: Listado básico de los objetos del sistema.

Esta lista no simboliza los únicos objetos disponibles, sinó que sirve de guía para los jugadores sobre los objetos más comunes que pueden comprar y vender con precios. Este listado está basado en los objetos más comunes en los sistemas de rol en los que se basa Animaia. Tanto los jugadores como el GM pueden ampliar o modificar la lista de objetos a voluntad si así lo desean.

5.2.6.3. Monedero y la economía monetaria

Cada personaje posee un monedero, que forma parte de su inventario donde guardar su dinero. En el mundo de Animaia la economía está basada en un sistema de 4 tipos diferentes de monedas. Estas son, de más a menos valor, monedas de paladio, de platino, de oro y de plata. Cada moneda vale 10 veces menos que la anterior:

1 moneda de paladio equivale a 10 monedas de platino o 100 de oro o 1000 de plata. Así funcionan las equivalencias de las monedas:

Moneda	1 Paladio	1 Platino	1 Oro	1 Plata
Paladio	1	0.1	0.01	0.001
Platino	10	1	0.1	0.01
Oro	100	10	1	0.1
Plata	1000	100	10	1

Figura 5.2.6.3/1: Tabla de equivalencias de las monedas del mundo de Animaia.

Estas equivalencias son lineales para mayor comprensión del jugador, y el uso de 4 monedas viene dado por el tema de diseño general del número 4. Dicho monedero está separado en 4 partes para organizar esas monedas de forma ordenada. Los precios del mundo siempre están estipulados por monedas de plata. Un símil con la vida real podría ser el uso de billetes, los cuales representan cantidades mayores al valor simple de la moneda. Una moneda de platino sería el equivalente a un billete de 100€. Estas monedas se pueden usar para intercambiar bienes y servicios, o sea, para comprar objetos en tiendas o tabernas o para poder usar alojamiento en un hostal, por ejemplo. Hay objetos que se pueden comprar y son de vital importancia para continuar la aventura, como pueden ser la comida y la bebida, así que las monedas son un recurso importante para los personajes. Es por eso que se debe motivar a los jugadores a completar misiones para las recompensas económicas. De forma predeterminada, todos los personajes creados siempre tienen una moneda de platino al iniciar la aventura, pero

eso se puede modificar para adecuar sus riquezas con sus posibles trasfondos.

5.2.7. Habilidades

El planeta de Animaia otorga a sus seres vivos poderes sobrenaturales, habilidades mágicas que pueden hacer gracias a la manipulación almática. Esta es la razón que se da para justificar la existencia de habilidades sobrenaturales en este sistema y que son típicas de los juegos de rol.

Para poder ejecutar esas habilidades, primero las deben memorizar. Eso se traduce a que los personajes sólo pueden aprender cierto número de habilidades según su valor Memoria. Cuanto más alto, más habilidades sabrán ejecutar. A partir de 6 de Memoria un personaje tendrá disponible un hueco de habilidad con el que memorizarlo. Los huecos se desbloquean cuando el valor Memoria alcanza un número par (8, 10, 12, 14...) hasta el máximo de 20, que permite al personaje memorizar hasta 8 habilidades. Esto limita a los personajes porque les obliga a elegir cuales son las habilidades que más les interesa para poder usarlas. Si quieren aprender más habilidades, deberán invertir puntos al valor Memoria en el momento de subir de nivel. Este sistema está inspirado en el sistema de huecos de hechizos de la saga Dark Souls.

Las habilidades disponibles de un personaje dependerá de la clase a la que pertenezcan. Esto sirve para limitar el estilo de juego de dicha clase y a que el jugador siga dicho estilo de juego de una forma más controlada.

El diseño de las habilidades ha seguido de cierta forma el diseño por sustracción, acuñado por Fumito Ueda. Teniendo en cuenta que el objetivo de este sistema de rol es la sencillez, crear un sistema de habilidades con complejidades como áreas de efecto, posibilidades de acierto o efectividades elementales, elementos típicos de los sistemas de rol más populares, resulta redundante. Es por eso que las habilidades de este sistema, al usarse se aplican directamente al objetivo elegido por el jugador, por lo tanto, no pueden fallar ni existen las efectividades elementales. Las habilidades solo pueden hacer 6 cosas. Hacer daño a un enemigo, hacer daño a varios enemigos,

curar a un aliado, mejorar algún o varios valores de uno a todos los aliados, desmejorar algún valor o varios valores de uno a todos los enemigos, o aplicar estados beneficiosos a un aliado.

Toda habilidad tiene un coste, que es fijo. Ese coste es de PA, que es el símil de este sistema al maná de otros sistemas de rol. El usuario puede usar cualquier habilidad que haya aprendido independientemente de si el coste supera a los Puntos de Alma disponibles, pero el jugador deberá tener en cuenta que si decide gastar todos sus Puntos de Alma restantes quedará exánime, por lo tanto, fuera de combate. Ese detalle sirve para que los jugadores puedan arriesgarse a usar una habilidad a la desesperada para terminar con sus enemigos o para que entiendan que el abuso de habilidades por encima de sus capacidades comportan consecuencias severas. También esta mecánica sirve como guiño al personaje Megumin perteneciente a la serie de novelas ligeras Kono Subarashii Sekai ni Shukufuku wo!, más conocida de manera abreviada como KonoSuba!

Si la habilidad tiene una efectividad, ya sea daño o curación entre otros efectos positivos, dicha habilidad tendrá dados vinculados, y su efectividad se calculará a partir de una tirada de dichos dados más el bono de Poder del personaje. Por ejemplo, el daño de la habilidad se calcula a partir de esta ecuación: (nº de dados del personaje x d6) + bono de Poder.

Si la habilidad se basa en un ataque especial con el arma, entonces esa habilidad calculará el daño a partir del dado vinculado al arma del personaje y al bono de valor vinculado a esa misma arma.

Toda habilidad puede usarse independientemente de si el personaje está alejado del enemigo o enfrentado y su objetivo puede ser cualquiera, tanto aliado como enemigo, siempre y cuando forme parte del conflicto.

5.2.8. Progresión

Como en casi todo sistema de rol, todos los personajes tienen una forma de mejorar sus estadísticas paulatinamente, mediante la experiencia y las subidas de nivel. Este sistema permite reflejar los esfuerzos y experiencias obtenidas por el personaje como el catalizador por el cual mejoran sus aptitudes.

En cada partida, según las acciones y el rendimiento en combate del personaje, se le otorgan puntos de experiencia. Estos sirven para poder subir de nivel. Subir de nivel simboliza la mejora del personaje, su progresión durante la campaña gracias a superar las dificultades que se enfrenta y le permite poder enfrentar desafíos más difíciles en un futuro. A cada nivel que suba el personaje, sus valores aumentarán de tal forma:

- VIDA: +8 por nivel (+2 por bono de Resistencia).
- ALMA: +8 por nivel (+2 por bono de Poder).
- +1 dado de ataque (Solo si alcanza un nivel par)
- +2 en el Atributo de clase.
- +1 en un Atributo de la raza a elección.
- +1 en un Atributo cualquiera a elección.

Para subir de nivel, se necesita llegar a ciertos números. Un personaje de nivel 1 necesita 100 puntos de experiencia para alcanzar el nivel 2. Estos puntos se acumulan, y el siguiente límite de nivel pasa a ser el doble más la mitad del límite anterior (L*2,5). Por ejemplo: De nivel 1 a 2 hay que conseguir 100 puntos, pero para llegar a nivel 3, hay que acumular (100*2,5) = 250 puntos de experiencia.

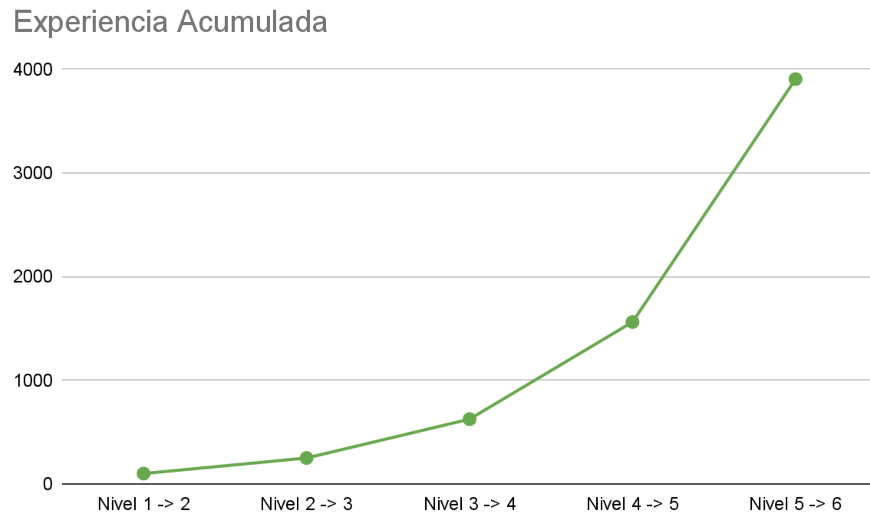

Figura 5.2.8/1: Gráfico de la experiencia necesaria para subir cada nivel hasta nivel 6.

Como esta fórmula es exponencial, a cada nivel, el GM tendrá que repartir más puntos de experiencia para los personajes, simbolizando que los desafíos que superan cada vez son más complicados, pero sobre todo para evitar la sensación de estancamiento en la progresión del personaje.

Como guía para repartir puntos de forma equilibrada al final de cada partida y para cada nivel, hay que dar los puntos necesarios para poder subir en 2, 3, 4, 5, 6... partidas dependiendo del nivel. O sea, para un personaje nivel 1, entre 40 y 60 puntos para las 2 primeras partidas están bien. Para un personaje de nivel 2, que sean necesarias entre 2 y 3 partidas para subir de nivel, para los de nivel 3, que necesiten entre 3 y 4 partidas... Es necesario provocar la sensación de dificultad mayor para subir de nivel mediante la subida de nivel cada vez menos frecuente, pero también hay que evitar que la ganancia de puntos sea lineal.

Para justificar las reparticiones de puntos, se recomienda otorgar puntos según los enemigos derrotados en combate, acciones acertadas para seguir la campaña de forma ideal, la interpretación del personaje hecha por su jugador, o sea, que su

interpretación sea sólida y tenga sentido durante la partida según como se ha descrito su personaje y su personalidad, o que hayan conseguido milagros en las tiradas durante la partida.

No se tiene que seguir a rajatabla esta repartición, el GM es libre de repartir los puntos que crea que son necesarios para cada personaje según su interpretación de la partida. Los jugadores pueden incluso negociar con el GM por un número razonable de puntos de experiencia, recordando siempre lo que han conseguido hacer durante la partida.

Al subir de nivel, el valor vinculado a la clase aumenta en 2 puntos automáticamente para representar la mejoría en su característica más importante. Luego el jugador puede elegir subir en un punto uno de los tres valores vinculados a la raza del personaje para simbolizar una mejoría por su naturaleza innata y finalmente se da a elegir un valor adicional, físico o mental, para aumentarlo también en un punto, siendo esta última elección útil para ofrecer más flexibilidad en el momento de mejorar el personaje y una mejoría general más notable entre niveles. El proceso de subida de nivel puede verse en forma de diagrama en la Figura 5.2.8/2.

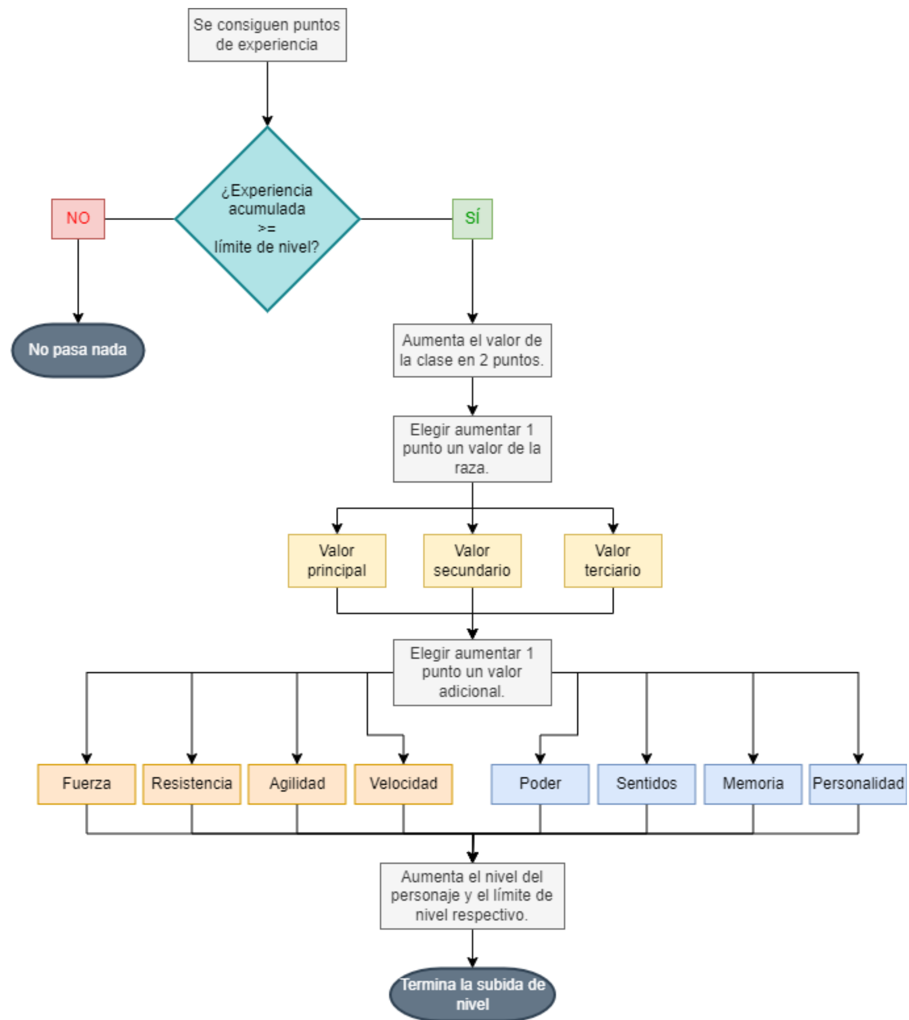

Figura 5.2.8/2: Diagrama de flujo de la subida de nivel.

5.2.9. Acciones

Interpretar a un personaje implica tomar control del mismo y actuar en el juego a través del mismo personaje. Eso quiere decir que los jugadores pueden hacer que sus personajes actúen para que encarnen las decisiones que toman durante las partidas. Hay diferentes acciones que pueden ejecutar los personajes en cualquier momento durante la partida, cada acción está vinculada a uno de los valores del personaje. Si el jugador quiere ejecutar una acción que no esté especificada en el cuadro, el GM tendrá que interpretarla y vincularla a uno de los valores del personaje para poder aplicar posibles bonos y así ajustar las posibilidades de éxito de esa acción según las competencia del personaje en esa acción. Los jugadores son libres de intentar hacer las acciones que quieran, pero el

sistema ofrece acciones predeterminadas que se prevé que sean las acciones más comunes durante las partidas. Estas son:

- **Percibir:** Vinculada al valor Sentidos. Simboliza la acción de mirar a algo, vigilar, inspeccionar con la mirada o buscar algo por el entorno. Todo lo que sea usar la visión para algo en específico entra dentro de esta acción.
- **Escuchar:** Vinculada al valor Sentidos. Simboliza la acción de usar el oído para tratar de oír algo, ya sean sonidos sospechosos, conversaciones...
- **Degustar:** Vinculada al valor Sentidos. Simboliza la acción de usar el sentido del gusto para poder detectar posibles trazas de ciertos elementos en comidas, bebidas o para identificar objetos.
- **Oler:** Vinculada al valor Sentidos. Simboliza la acción de usar el olfato para poder detectar rastros de olores y así poder identificar trazas de ciertos elementos en cualquier cosa.
- **Correr:** Vinculada al valor Velocidad. Simboliza la acción de correr o esprintar una cierta distancia o durante cierto tiempo ya sea durante una persecución o una competición.
- **Saltar:** Vinculada al valor Fuerza. Simboliza la acción de saltar vertical o longitudinal una cierta distancia para poder superar un obstáculo, alcanzar algo en lo alto o caer sin hacerse daño.
- **Trepar:** Vinculada al valor Agilidad. Simboliza la acción de subir o bajar por una superficie vertical ya sea una pared, un acantilado, un árbol...
- **Nadar:** Vinculada al valor Resistencia. Simboliza la acción de moverse dentro del agua, incluyendo también el buceo.
- **Persuadir:** Vinculada al valor Personalidad. Simboliza la acción de conversar con alguien para conseguir con razones y argumentos que actúe o piense de un modo determinado. Tranquilizar, encandilar o seducir son consideradas como técnicas de persuasión.

- **Intimidar:** Vinculada al valor Fuerza. Simboliza la acción de hacer que alguien sienta miedo o temor para extorsionar o asustar.
- **Negociar:** Vinculada al valor Personalidad. Simboliza la acción de hablar de un asunto para llegar a un acuerdo o solución.
- **Mentir:** Vinculada al valor Personalidad. Simboliza la acción de decir deliberadamente lo contrario de lo que se sabe, se cree o se piensa que es verdad con el fin de engañar a alguien.
- **Robar:** Vinculada al valor Agilidad. Simboliza la acción de quitar a una persona algo que le pertenece de forma que no se dé cuenta.
- **Sigilar:** Vinculada al valor Agilidad. Simboliza la acción de mantenerse oculto o moverse sin ser detectado.
- **Disfrazar:** Vinculada al valor Agilidad. Simboliza la acción de modificar de alguna forma la apariencia de algo o alguien usando ropa, máscaras y/o maquillaje con el fin de ocultar su verdadera identidad.
- **Recordar:** Vinculada al valor Memoria. Simboliza la acción de traer a la memoria propia algo percibido, aprendido o conocido, o retener algo en la mente.
- **Magicología:** Vinculada al valor Poder. Simboliza la acción de usar los propios conocimientos de la magia para detectar rastros de su uso o reconocer magias entre otras acciones que se relacionen con la magia y su historia.
- **Medicina:** Vinculada al valor Resistencia. Simboliza la acción de usar los propios conocimientos sobre la medicina y antropología para poder reconocer dolencias, síntomas y lesiones o para curar a alguien.
- **Historia:** Vinculada al valor Memoria. Simboliza la acción de usar los propios conocimientos sobre la historia del mundo para recordar acontecimientos del pasado o verificar la autenticidad y valor de un objeto.
- **Legislación:** Vinculada al valor Memoria. Simboliza la acción de usar los propios conocimientos de las leyes

para poder defender derechos o aprovechar huecos legales.

- **Zoología:** Vinculada al valor Memoria. Simboliza la acción de usar los propios conocimientos sobre zoología para identificar animales ya sea viéndolos directamente o por rastros dejados por estos como pisadas, excrementos, etcétera.
- **Botánica:** Vinculada al valor Memoria. Simboliza la acción de usar los propios conocimientos sobre botánica para identificar todo tipo de plantas y vegetación según su follaje, tipo de tierra en la que crecen, frutos, tallo, clima en el que se encuentran, etcétera.
- **Ciencia:** Vinculada al valor Memoria. Simboliza la acción de usar los propios conocimientos sobre cualquier otra ciencia no especificada anteriormente como física, química, astronomía, biología... Para detectar o crear algún químico, aprovechar fenómenos de la física, guiarse por las estrellas, etcétera.
- **Bailar:** Vinculada al valor Agilidad. Simboliza la acción de mover el cuerpo al ritmo de la música.
- **Cantar:** Vinculada al valor Personalidad. Simboliza la acción de crear una melodía musical usando la voz o haciendo sonidos con la boca.

En el caso de que los jugadores pidan ejecutar una acción que no está especificada en esta lista, el GM tendrá que improvisar para adaptar la acción al sistema, vinculando uno de los atributos del personaje a dicha acción. También dependerá del contexto de la acción. Por ejemplo, si un jugador pide forzar una cerradura, el GM necesitará contexto sobre cómo quiere el jugador forzar esa cerradura para saber que tipo de atributo tendrá que vincular. Puede que el jugador quiera aprovechar su fuerza para romper la cerradura a lo bruto, o puede que tenga una ganzúa y decida usar sus habilidades para abrir la cerradura. En el primer caso, claramente el jugador pide utilizar su valor de Fuerza para forzar la cerradura, pero en el segundo caso, lo idóneo es vincular la Agilidad a la acción.

Para ejecutarse, el jugador tendrá que tirar un dado d20 para conseguir el resultado de la acción. A continuación tendrá que sumar o restar el bono el valor vinculado a dicha acción para conseguir el resultado final.

Para interpretar el resultado de la acción, se compara el resultado con la dificultad de la acción. Si el resultado es superior, se considera un éxito y se permite hacer la acción y sus consecuencias. De lo contrario, se interpreta el resultado como un intento fallido de ejecutar la acción. Todo el proceso de la ejecución de una acción puede verse en forma de diagrama en la Figura 5.2.9/1

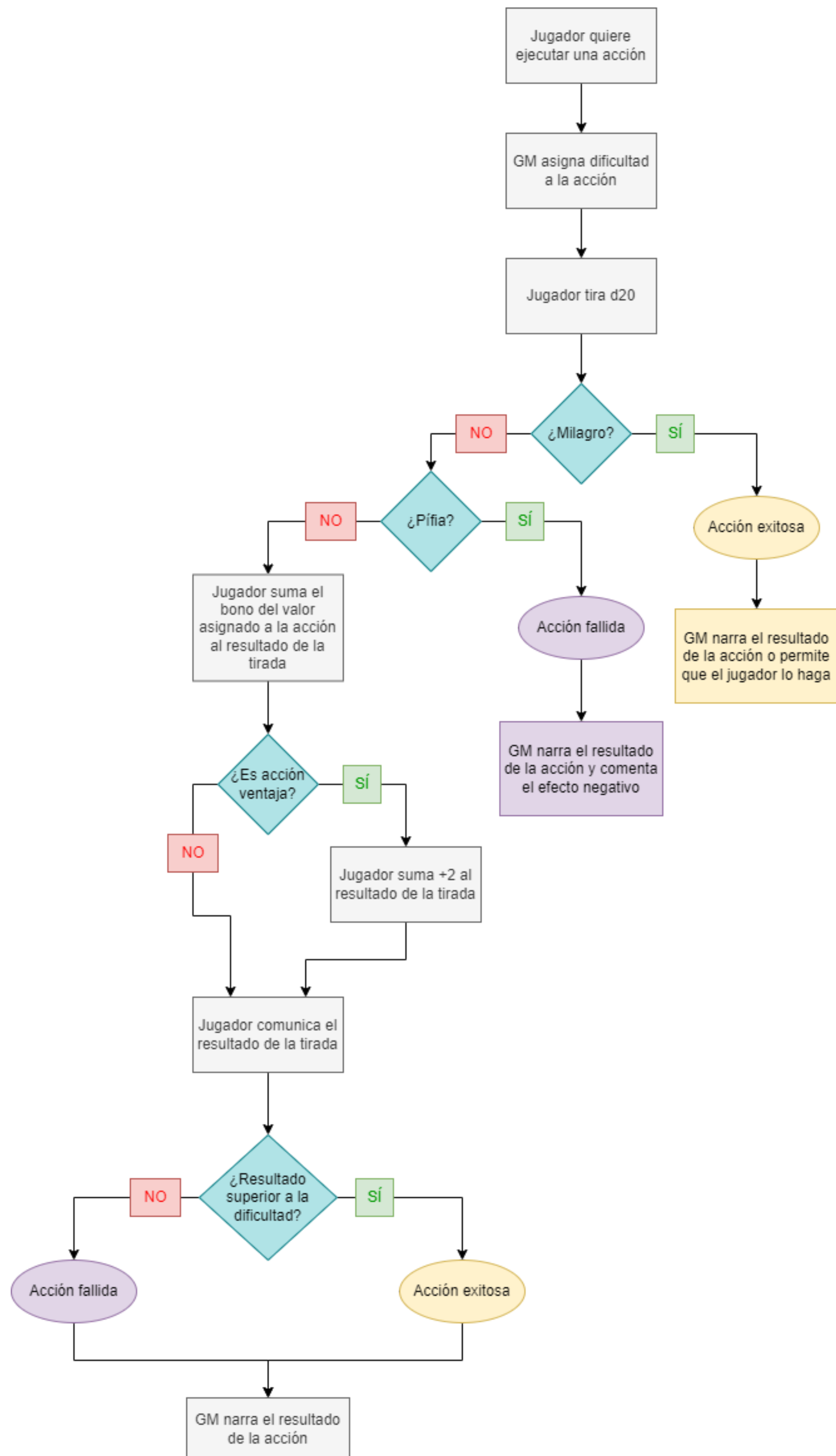

Figura 5.2.9/1: Diagrama de flujo de la ejecución de una acción.

5.2.9.1. Dificultades

Muchas acciones deben suponer un reto para el personaje que las ejecute. Para eso, el GM debe asignar una dificultad adecuada según la acción que se quiera

hacer y el contexto de la acción. Esta mecánica sirve para añadir un elemento de aleatoriedad que permite la generación de situaciones inesperadas e interesantes para los jugadores, manteniendo el estado de flow en ellos durante las fases de interpretación.

A continuación está la lista de las dificultades. Esta tiene en cuenta que la mayor puntuación base es un 20 por usarse un dado d20, pero que puede ser superada por diversos bonos del personaje.

- **Ordinaria:** 8 o superior. No son particularmente difíciles, pero con un poco de mala suerte o no siendo proficientes en ese tipo de acción, puede llevar al fallo de esta.
- **Compleja:** 16 o superior. Acciones que requieren de una buena habilidad por parte del personaje o de bastante suerte. Si el personaje es aventajado en el atributo de la acción, no debería encontrar dificultad a la hora de ejecutarla.
- **Sobrehumana:** 24 o superior. Acciones que para una persona común sería totalmente imposible, y que para alguien quien se especializa en el atributo de la acción requiere esfuerzo y un poquito de suerte para realizarla con éxito.
- **Cénit:** 32 o superior. Conseguir superar estas acciones requiere del mayor esfuerzo de un personaje especialista. Conseguirlo significa llegar a los límites de lo que es posible, solo reservado para aquellos quienes representan el culmen del mundo en esas habilidades.
- **Divina:** 40 o superior. Acciones reservadas solo para aquellos que se escapan de las ataduras terrenales, que sus poderes están fuera de la comprensión y la lógica razonable. Si el jugador consigue un milagro en la tirada del d20, esto no garantizará el éxito de la acción. Solo podrán ser

ejecutadas si el personaje está capacitado de forma sobrenatural a través de mejoras obtenidas por diversas fuentes.

5.2.9.2. Pifias y milagros

Esta es una mecánica común de los sistemas de rol, donde los jugadores pueden provocar un fallo asegurado que conlleve consecuencias negativas o un acierto asegurado con efectos positivos. Esto existe para añadir un elemento más de tensión a los jugadores, haciendo más interesantes las tiradas de dados.

Una pifia ocurre cuando se consigue un 1 con el d20. Si esto ocurre, la acción fallará independientemente del resultado total con los bonos, y si el GM interpreta necesario, provocará un efecto negativo (daño a sí mismo, alertar enemigos,...).

Un milagro ocurre cuando se consigue un 20 con el d20. Si esto ocurre, la acción será exitosa independientemente de su dificultad, y si el GM interpreta necesario, provocará un efecto positivo o permitirá al jugador interpretar cómo quiera como ocurre la acción a placer.

5.2.9.3. Acciones ventaja

Cada personaje tiene 4 acciones con las que sus tiradas tienen un aumento de 2 puntos en sus resultados. En el momento de crear el personaje, el jugador debe seleccionar las acciones predeterminadas con las que quiera que su personaje sea más capacitado. Esto permite a los jugadores que sus personajes se especialicen en unas acciones que no se tengan que basar en los valores de dicho personaje.

5.2.9.1. Interacciones

La interacción es una de las piezas claves en el momento de crear una experiencia inmersiva para los jugadores, por ende, es crucial que los jugadores puedan interactuar con cualquier cosa y que el GM les describa el resultado de dichas interacciones de forma detallada. Por ejemplo,

si alguien quiere abrir una puerta, puede que lo intente usando una ganzúa, o quiere aprovechar su fuerza corporal para hacerlo a lo bruto. El trabajo del GM es permitir que los jugadores puedan tomar estas alternativas usando los Valores más idóneos por el tipo de acción, asignarles una dificultad de éxito a dichas acciones y dependiendo del resultado, hacer una descripción adecuada de las consecuencias de esas acciones.

Durante la exploración de una zona, el GM debe dar una descripción detallada del lugar y dejar que los jugadores la exploren por su cuenta, permitiendo interactuar con cualquier cosa que quieran y de la forma que quieran. A veces, las acciones de un personaje pueden provocar consecuencias que cambien la situación mucho, como puede ser por ejemplo alertar enemigos o hacerlos aparecer para empezar un combate; o puede ocurrir que descubran un pasaje secreto que les lleve a un tesoro o incluso provocar un derrumbamiento que divida el grupo. Esas consecuencias son a libre decisión del GM.

Durante la campaña, los personajes también podrán interactuar con los habitantes de este mundo, a poder ser manteniendo su "role play" de sus personajes. Este tipo de interacción puede ocurrir por varias razones, ya sea para recopilar información de un lugar, intentar convencer a alguien de algo o simplemente para comprar cualquier cosa a algún mercader. El GM se encarga de interpretar a cualquier personaje que no sea controlable por los jugadores, también llamados PNJ (Personaje No Jugable), llevando a cabo las acciones que harían estos personajes. La interpretación, como siempre, depende de la situación. Normalmente, todos los PNJ serán amistosos o neutrales con los personajes de los jugadores, pero también pueden ser hostiles hacia ellos. Los PNJ amistosos tratarán de ayudar a los personajes de los jugadores, mientras que los hostiles intentarán

dificultar sus aventuras, ya sea entablando combate o por otros medios. La relación de los PNJ hacia los personajes de los jugadores puede variar según las acciones que ocurran entre ellos; por ejemplo, un PNJ neutral puede volverse hostil si es provocado o los jugadores hacen algo que pueda causar su enfado. Lo mismo puede ocurrir al revés, los jugadores pueden intentar apaciguar el conflicto con un PNJ hostil y lograr que se vuelva simplemente neutral o incluso amistoso. Todo eso queda a la interpretación del GM, quién decidirá la personalidad de los PNJ en todo momento y de cómo reaccionan a diversas situaciones.

Si un jugador intenta hacer algo que no sea atacar hacia un PNJ, el GM deberá pedirle que haga una tirada de la acción que represente de la mejor forma las intenciones del jugador. El GM entonces debe hacer también una tirada de la misma acción, pero teniendo en cuenta los Valores del PNJ. Si la tirada del jugador es superior, la acción es un éxito, pero de lo contrario, la acción falla. En caso de haber un empate, la acción sigue siendo exitosa. Esto sirve para influenciar en el tipo de acciones que frecuentan cada jugador acorde a los Valores de su personaje para incentivar un "role play" lógico.

El sistema de interacciones, como se ha comentado, es muy subjetivo, y depende del contexto del momento y la interpretación de los jugadores y GM, no sigue un orden completamente establecido debido a la libertad inherente del sistema de rol.

5.2.9.2. Comer y Beber

Aunque este juego se base en un mundo de fantasía, los personajes de los jugadores también necesitan alimentarse para seguir vivos. Comer y beber es esencial para mantenerse saludables, y olvidarse de algo tan

importante como esto puede acarrear condiciones graves que pueden llevar a la muerte.

Cada personaje necesita comer una ración de comida y beber 2 litros de líquido al día para mantenerse en forma, y no cumplir con estos requisitos trae consecuencias. Por cada día que un personaje no haya comido o bebido, todos sus Valores recibirán una penalización de -1. En caso de no comer ni beber en todo un día, se aplicará una penalización de -2 a todos sus Valores. Al cabo de 4 días sin comer ni beber, el personaje muere. Ver **Figura 5.2.9.2/1**.

Volver a comer y beber adecuadamente reinicia toda penalización por falta de comida, pero solo cuando se cumplen ambas cosas. Si solo se come o se bebe apropiadamente después de un día sin hacerlo no habrá penalización extra, pero se mantendrá la actual.

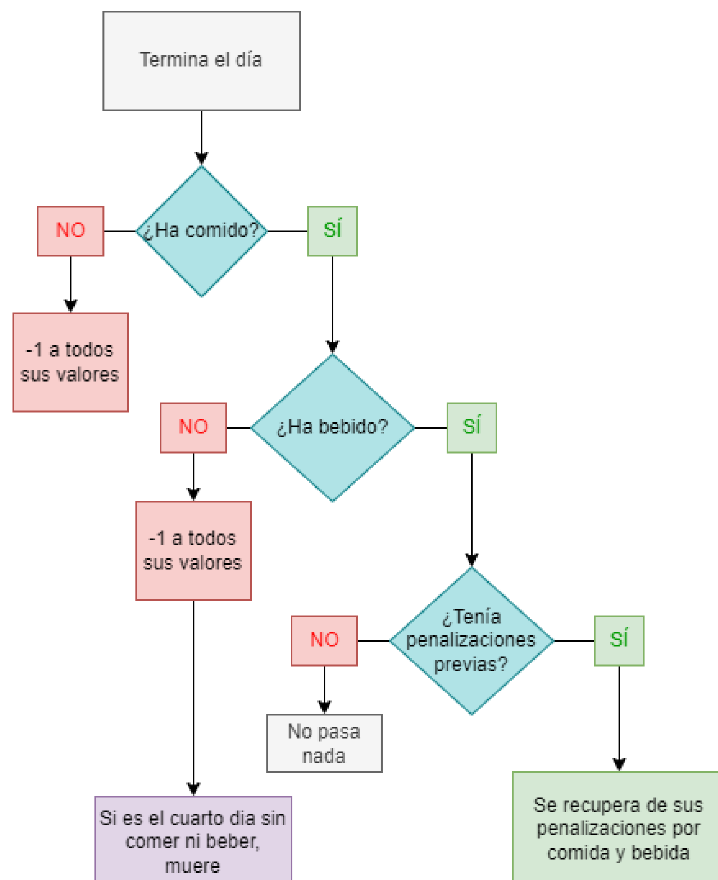

Figura 5.2.9.2/1: Diagrama de flujo del sistema de comida y bebida.

La existencia de esta mecánica está pensada para dar utilidad a los objetos de comida y bebida, además de humanizar más a los personajes para que los jugadores se vinculen y se preocupen más por ellos. También existe para que los jugadores planeen bien sus viajes, obligándolos a tomar las provisiones necesarias para que no les afecte las secuelas de no comer ni beber.

5.2.9.3. Descanso

El descanso sirve para recuperar PC y PA dependiendo de las horas descansadas. Se recupera un 10% de los PC y PA máximos por hora.

Descansar no es sinónimo de dormir. Descansar en este sistema simboliza la acción de evitar la actividad física y relajarse durante un rato determinado. Se puede leer algún libro o conversar tranquilamente mientras se descansa. Dormir también está entre las muchas actividades relajantes que se pueden hacer.

No descansar a partir del segundo día sin hacerlo conlleva a una penalización de -2 en todos los Valores del personaje. Al cabo de 4 días sin descanso, el personaje muere.

Sólo descansar 8 horas permite al personaje recuperarse completamente de las penalizaciones por falta de descanso. Descansar menos de esas horas en todo el día solo recupera +1 en los Valores del personaje. Ver **Figura 5.2.9.3/1**.

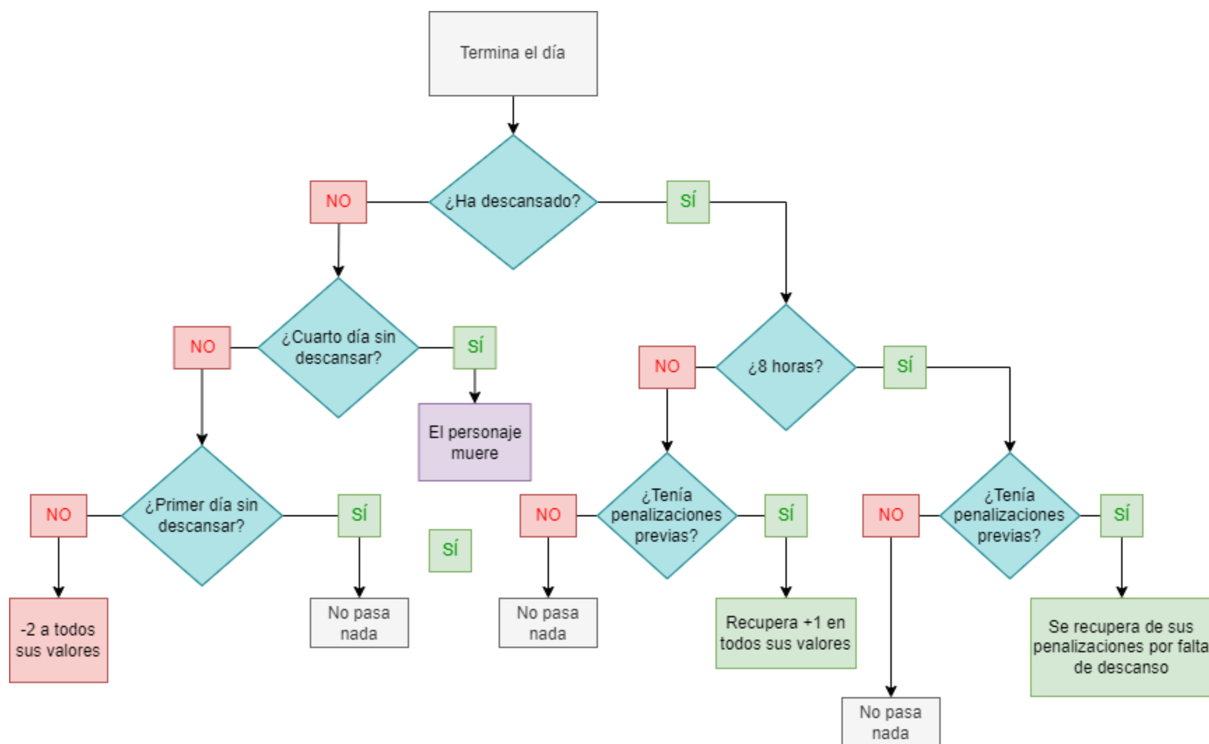

Figura 5.2.9.3/1: Diagrama de flujo del sistema de descansos.

Esta mecánica se usa principalmente como método de recuperación de los PC y PA del personaje. También, igual que la mecánica de la comida y la bebida, ayuda a humanizar a los personajes y así apoyar y reforzar el sentimiento de apego de los jugadores por sus personajes. Además, ofrece la oportunidad a los jugadores de desarrollar a los personajes ante sus compañeros durante los periodos de descanso.

5.2.10. Combate

El sistema de combate está basado en un sistema de turnos, donde cada personaje puede ejecutar un número específico de acciones determinado por sus APT(Acciones por Turno). En un combate toman parte los personajes que controla cada jugador y los enemigos, que los controla el GM.

El diseño del sistema de combate sigue el diseño por sustracción. Muchos aspectos típicos del combate de los sistemas en los que se basa Animaia han sido omitidos o

modificados para ofrecer una experiencia más simple y comprensible.

Un combate empieza cuando un personaje del GM o un jugador decide usar una acción especial de combate sobre alguien que puede acarrear una respuesta violenta. La decisión de iniciar el combate siempre la tomará el GM interpretando la situación. Para que se inicie el combate debe haber enemigos contra los que combatir, y una razón de peso para que se recurra a la violencia en la situación determinada.

Al iniciar un combate, el orden de los turnos se determina por el valor de Velocidad de cada personaje que entre en combate, tanto jugadores como enemigos, siendo el que tenga el valor más alto el primero, y ordenándose así hasta el personaje más lento, que será el último en actuar en un turno. En caso de que haya un empate entre personajes y/o enemigos, se hará una tirada de un d6 para el desempate. Quien saque el valor más alto tendrá su turno antes. También, al iniciar un combate, todos los personajes y enemigos empiezan en el estado Alejado (más adelante se explica que es este estado).

Puede ocurrir que un combate se inicie por un ataque sorpresa, provocando que el personaje atacado no pueda defenderse del ataque. Eso significa que el ataque ejecutado no tendrá fase de esquiva o bloqueo y automáticamente iniciará el combate, dejando al atacante y al atacado enfrentados solo si el ataque ha sido con un arma cuerpo a cuerpo. Ver **Figura 5.2.10/2.**

Todos los personajes involucrados, tanto jugadores como enemigos, seguirán el orden de turno determinado al inicio del conflicto. En sus turnos, los personajes pueden ejecutar un número de acciones determinado por su APT. Estas acciones pueden ser las que pueden hacer fuera de combate, pero a estas se les añaden unas acciones especiales de combate. Estas son: Enfrentarse, Alejarse, Atacar, Proteger, Esquivar y

Bloquear. Estas acciones están separadas en 2 categorías específicas, movimiento y ofensiva. Ver **Figura 5.2.10/3.**

Un combate termina cuando se llega a una conclusión pacífica ya sea por huír, negociar el alto a la violencia, rendición, perdonar a los enemigos bajos en vida...; o, por defecto, cuando todos los integrantes de un grupo terminan fuera de combate, o sea, cuando sus PC o PA acaban a 0. Al salir victoriosos de un combate, todos los integrantes del grupo conseguirán puntos de experiencia dependiendo de los enemigos derrotados que les servirá para subir de nivel. En caso de que algún personaje haya huído de combate o haya muerto, no se le otorgarán puntos de experiencia. En la **Figura 5.2.10/1** se puede observar un diagrama del proceso completo de un combate.

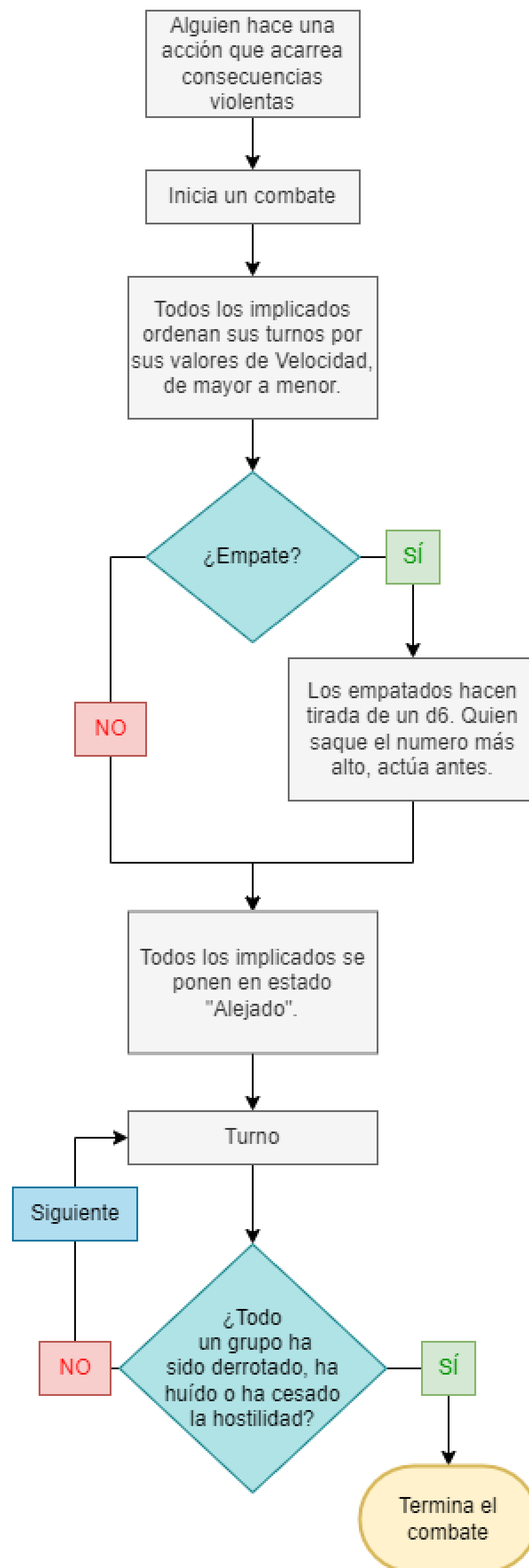

Figura 5.2.10/1: Diagrama de flujo general del combate

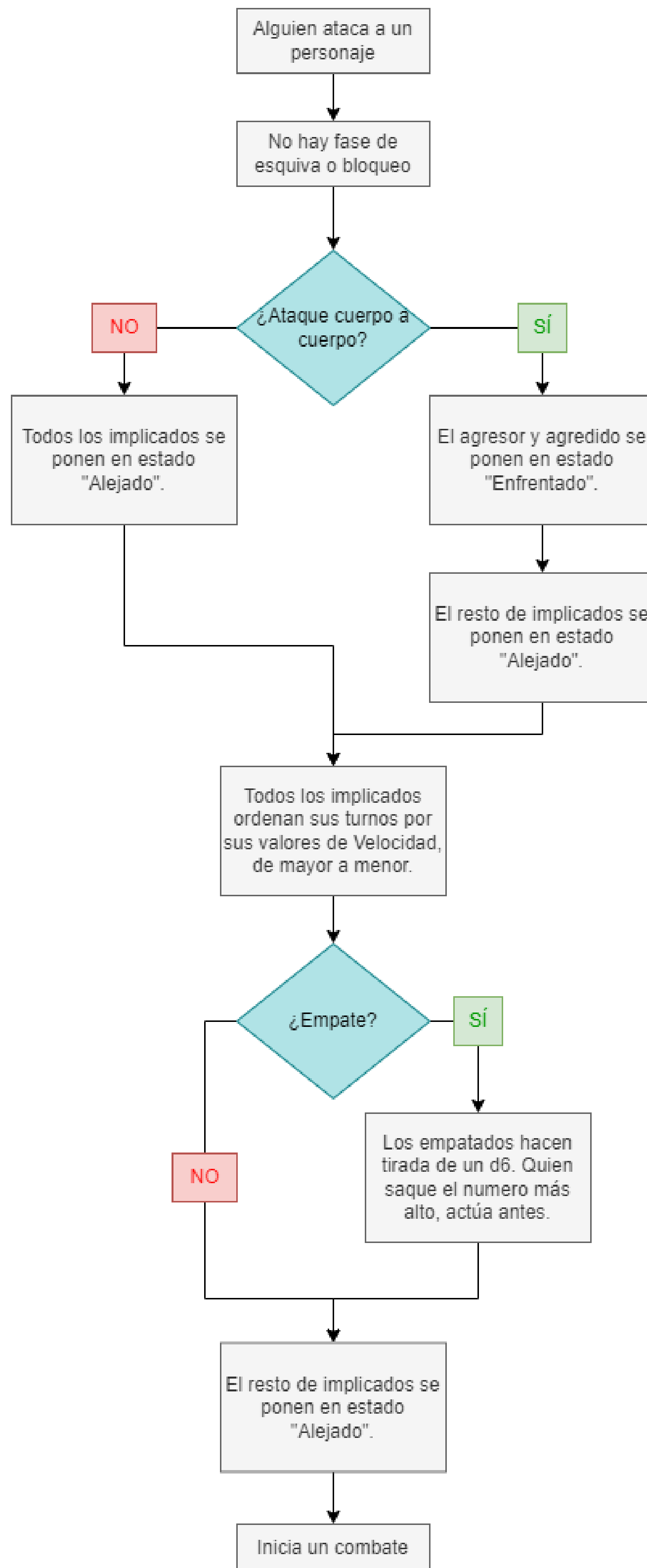

Figura 5.2.10/2: Diagrama de flujo del inicio de un combate por ataque sorpresa

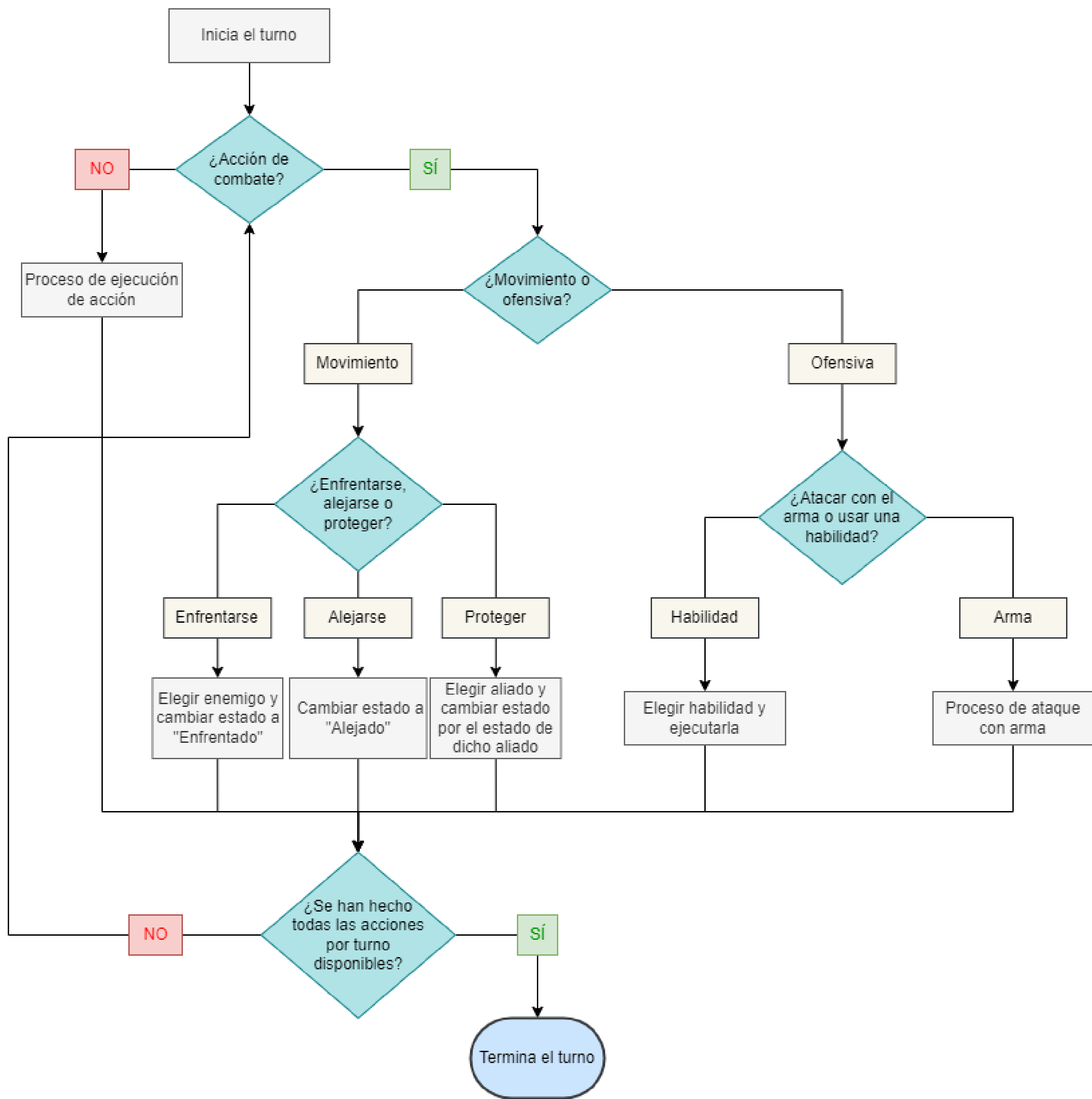

Figura 5.2.10/3: Diagrama de flujo del proceso general de un turno

5.2.10.1. Movimiento

Uno de los elementos simplificados, en lugar de tener en cuenta la posición y distancias sobre cada personaje, el posicionamiento y movimiento se limita a 3 acciones, que son exclusivas del combate. El uso de cada una de esas acciones cuenta como una acción del turno del personaje. Estas son:

- **Alejarse:** Simboliza la acción de posicionarse donde no se está al alcance de ataques enemigos cuerpo a cuerpo. Aún así, se puede ser atacado o atacar con ataques a distancia o habilidades. Al

hacer esta acción, se considera que está en el estado Alejado. Alejado significa que no tiene enemigos a rango de cuerpo a cuerpo, pero aun se puede recibir habilidades o ataques a distancia y se puede atacar a distancia.

- **Enfrentarse:** Simboliza la acción de posicionarse al alcance de los ataques cuerpo a cuerpo contra un enemigo. Suele usarse para preparar una ofensiva contra un enemigo. Mantenerse enfrentado significa estar al alcance de los ataques cuerpo a cuerpo enemigos. Al hacer esta acción, se considera que está en el estado Enfrentado. Enfrentado significa que tiene a uno o varios enemigos a rango de ataque cuerpo a cuerpo.
- **Proteger:** Con esta acción, un personaje se acerca a un aliado y se vuelve su protector. Si este aliado es atacado, se puede intentar bloquear los ataques que sean dirigidos en su lugar o atacar a los enemigos con los que está enfrentado.

5.2.10.2. Ofensiva.

El momento de atacar puede dividirse en dos partes. Cuando el atacante decide atacar, este primero escoge si agredir usando su arma o una habilidad ofensiva. Entonces se decide el poderío del ataque ejecutando la acción de Atacar. Si se quiere atacar con un arma cuerpo a cuerpo, el agresor debe estar enfrentado con el agredido. Con habilidades, armas a distancia o armas que pueden ser lanzadas no tienen esa restricción.

Cada ataque con el arma se calcula de la siguiente forma. Se hace una tirada de dados según el tipo de dado vinculado al arma que usa el personaje y el número de dados de ataque que posee. La suma del resultado de todos los dados lanzados más el bono del valor al que el arma está vinculado acaba siendo el daño total que ese ataque hará al objetivo.

Cuando el atacante sepa la potencia de su ataque y solo cuando ese ataque es con el arma, el objetivo del ataque tendrá que determinar cómo reacciona, ejecutando las acciones de Esquiva o Bloqueo. Cuando haya ejecutado su acción de reacción, el atacante hará lo mismo para determinar si el ataque consigue dañar al agredido o no. Estas son las acciones de reacción del sistema:

- **Esquiva:** Después de calcular el daño del ataque, el atacante y el atacado(s) tiran un d20 y suman su propio bono de Agilidad. Si la tirada del agresor es superior a la del agredido, el ataque acierta. Si la tirada del agresor es igual o inferior a la del agredido, el ataque es fallido y el agredido no recibe daño.
- **Bloqueo:** Después de calcular el daño del ataque, el atacante y el atacado tiran un d20. El agresor suma su resultado del d20 con el bono correspondiente del ataque (Fuerza para armas de Fuerza, Agilidad para armas de Agilidad) mientras que el agredido suma su resultado del d20 con su bono de Resistencia. Si la tirada del agredido supera a la del atacante, entonces la diferencia entre ellas serán los puntos de daño bloqueados por el defensor, por lo tanto, el daño infligido se restará a la diferencia obtenida. De esta forma es posible bloquear todo el daño de un ataque. Esta acción puede tomarla un personaje que esté protegiendo al objetivo original del ataque, pero no puede usarse por más de un personaje para el mismo ataque.

Estas acciones, aunque añaden una capa de complejidad al combate, lo convierte en algo más interesante por el simple hecho de que todos los participantes puedan reaccionar y decidir qué hacer ante los ataques, ofreciendo la oportunidad de poder evitar ser dañados

en vez de simplemente golpear y recibir golpes, cosa que puede llegar a ser monótona muy rápido.

Como en estas acciones se usa un dado d20 para decidir el resultado, también se puede aplicar el sistema de pifias y milagros de las acciones predeterminadas. En las figuras **5.2.10.2/1** y **5.2.10.2/2** se pueden observar los posibles resultados según las tiradas de dados.

ESQUIVA	Pifia	Tirada Normal	Milagro	ATACADO
Pifia	Ataque Fallido	Contraataque	Contraataque	
Tirada Normal	Daño Doble	Función Normal	Contraataque	
Milagro	Daño Doble	Daño Doble	Ataque Normal	
ATACANTE				

Figura 5.2.10.2/1: Tabla de resultados de los milagros y pifias durante una fase de esquiva.

BLOQUEO	Pifia	Tirada Normal	Milagro	ATACADO
Pifia	Ataque Fallido	Contraataque	Contraataque	
Tirada Normal	Daño Doble	Función Normal	Contraataque	
Milagro	Daño Doble	Daño Doble	Ataque Normal	
ATACANTE				

Figura 5.2.10.2/2: Tabla de resultados de los milagros y pifias durante una fase de bloqueo.

Función Normal: El resultado de la acción será determinado según lo explicado anteriormente.
Ataque Fallido: El ataque no hará daño.
Ataque Normal: El ataque dañará al atacado según el primer cálculo de daño.
Daño Doble: El ataque hará el doble de daño de lo calculado anteriormente.
Contraataque: Significa que el atacado evitará recibir daño y tendrá la oportunidad de ejecutar una acción en ese mismo turno.

Ver figuras **5.2.10.2/3, 5.2.10.2/4, 5.2.10.2/5** y **5.2.10.2/6** para observar el flujo del combate.

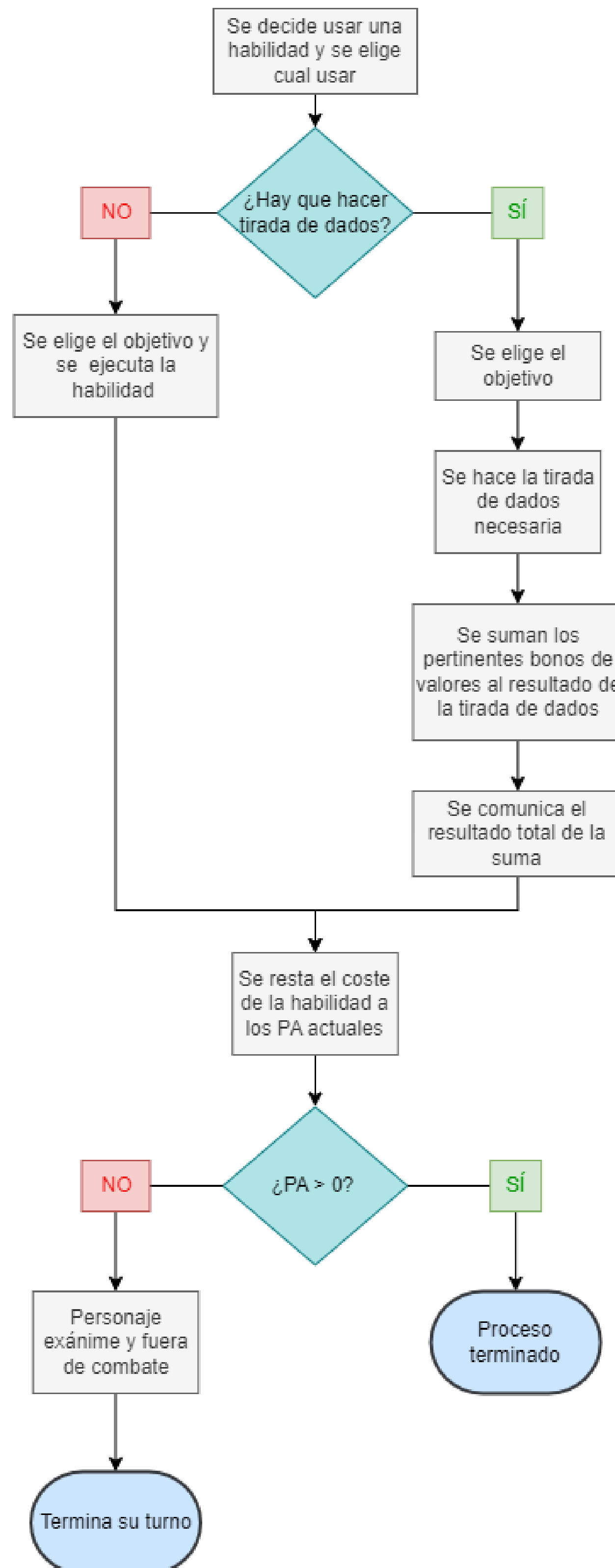

Figura 5.2.10.2/3: Diagrama de flujo del proceso del uso de una habilidad en combate.

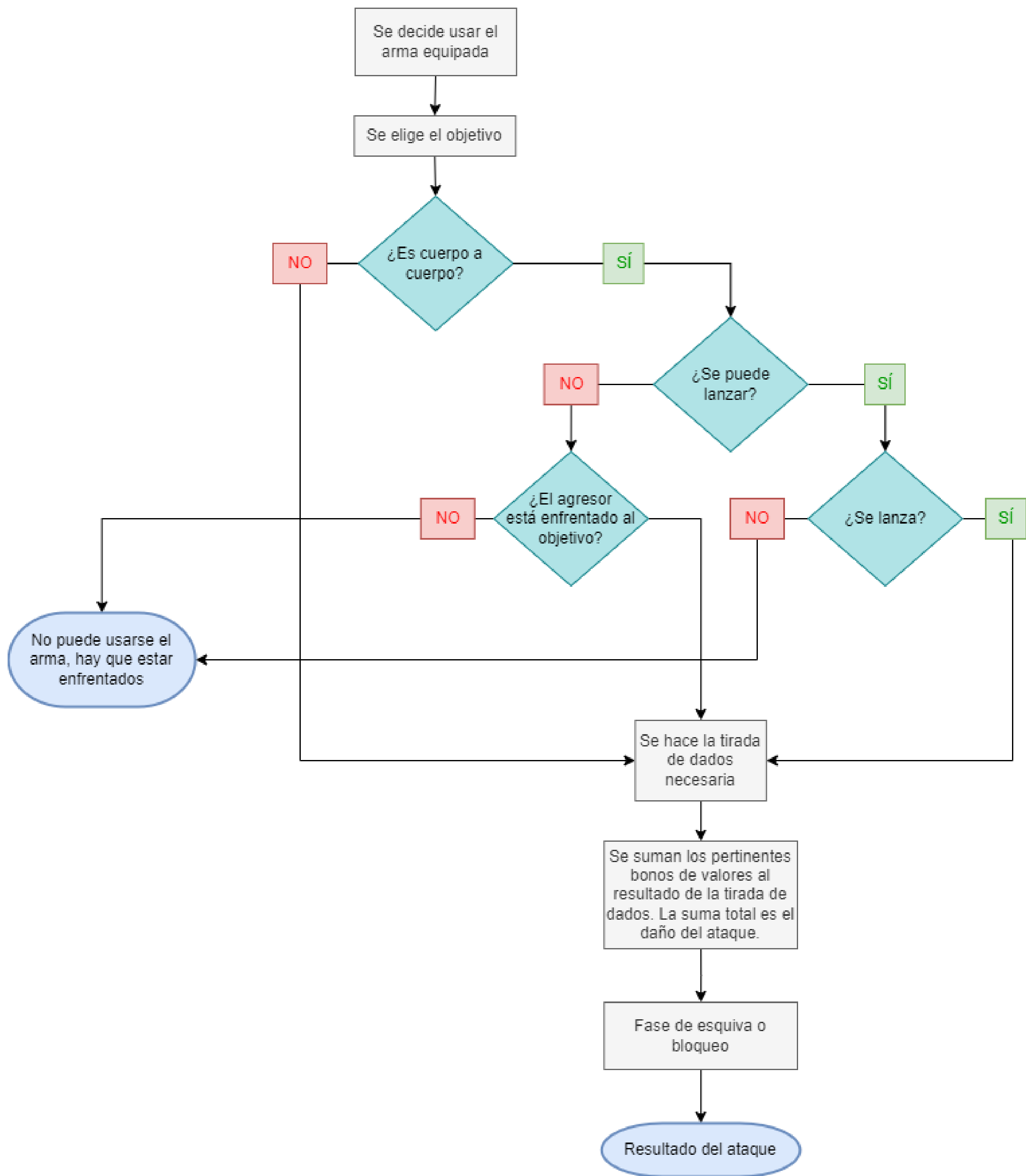

Figura 5.2.10.2/4: Diagrama de flujo del proceso de ataque con arma. Parte 1.

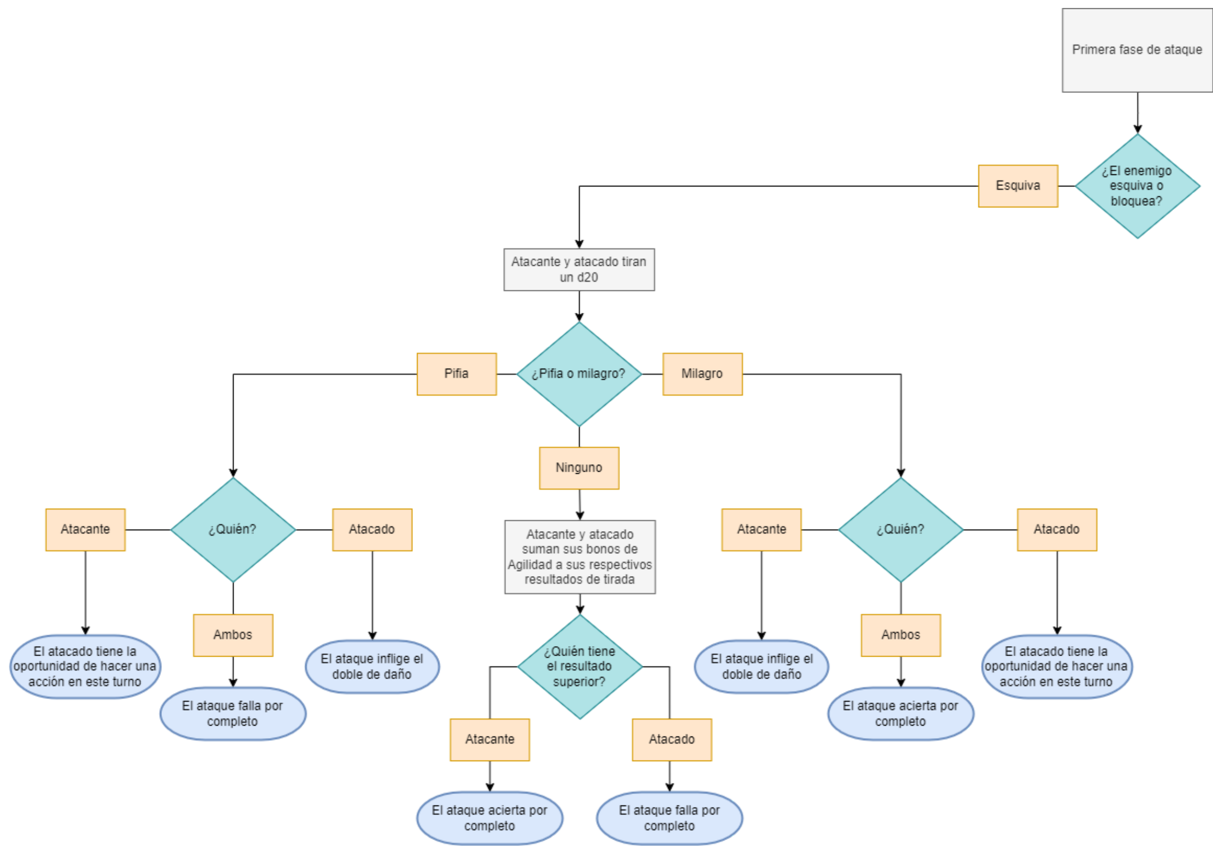

Figura 5.2.10.2/5: Diagrama de flujo del proceso de ataque con arma. Parte 2, fase de esquiva.

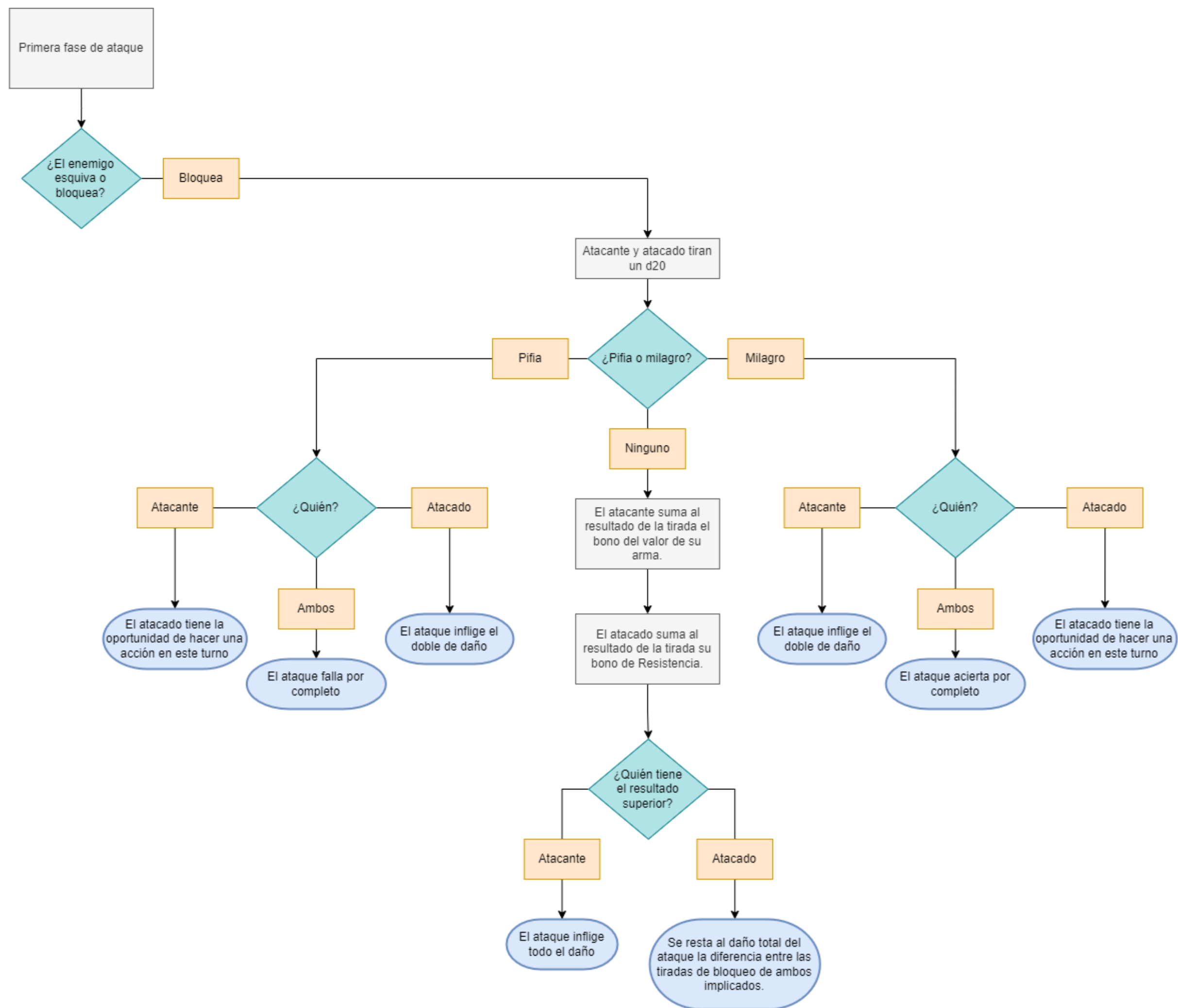

Figura 5.2.10.2/6: Diagrama de flujo del proceso de ataque con arma. Parte 3, fase de bloqueo.

5.2.10.3. Estados "Inconsciente", "Exánime" y "fuera de combate"

Los Puntos de Cuerpo(PC) y los Puntos de Alma(PA) son los recursos básicos que tiene un personaje durante el combate. Durante el combate esos recursos irán variando.

Los PC simbolizan la vitalidad o la vida del personaje, la integridad de su cuerpo. Cuanta más PC tenga el personaje, más robusto es, mientras que si tiene menos PC, más frágil es. Cuando este recibe daño, ese daño se resta a sus PC actuales. Esos PC pueden llegar a ser negativos; si eso ocurre, el personaje se desmayará y quedará fuera de combate. Si se llega a tener sus PC

actuales como sus PC máximos en negativo, ese personaje morirá.

Los PA simbolizan la cantidad de alma que tiene el personaje, la energía que necesita para poder seguir adelante, su voluntad. Los PA se usan para ejecutar habilidades, cuantos más PA tenga, el personaje podrá hacer habilidades más poderosas o más cantidad de ellas. Cuando un personaje realiza una habilidad, el coste de esa habilidad se resta a sus PA actuales. Los PA pueden llegar a estar en negativo si se usa una habilidad de coste mayor a los PA actuales, pero si eso ocurre, el personaje quedará exánime y fuera de combate.

Fuera de combate significa que el personaje no puede actuar hasta que se recupere. Solo puede recuperarse de 2 formas durante el combate, si algún aliado le cura para que no esté en puntos negativos de vida o alma, o esperando 3 turnos, en el que se recuperará naturalmente hasta quedarse a 1 punto de la estadística por la que ha acabado fuera de combate. Esto último solo pueden hacerlo los personajes de los jugadores, para dar un poco de flexibilidad a los jugadores. Si el combate termina en victoria de su grupo antes de que el personaje se pudiera recuperar, este recuperará la consciencia inmediatamente, quedándo a 1 punto de la estadística por la que ha acabado fuera de combate.

Aunque un personaje esté fuera de combate, no significa que no pueda ser atacado. Esto significa que si alguien decide intentar rematarlo, puede hacerlo, y si consigue dañarlo hasta igualar o superar su vida máxima en negativo significa la muerte de éste, y no va a poder volver a ser usado. Si un enemigo ataca a un personaje fuera de combate no se reiniciarán los turnos para recuperarse.

Para la interpretación del GM, los enemigos siempre priorizarán dejarlos a todos fuera de combate, y si lo consiguen, entonces empezarán a rematarlos. Por esto, es importante que los jugadores intenten proteger a sus compañeros fuera de combate, y si se ven abrumados, priorizar la huída antes que intentar acabar el combate. Si un personaje se recupera de este estado, podrá volver a actuar cuando le llegue su turno. También se recomienda que los enemigos controlados por el GM no ataquen al personaje que acaba de recuperarse hasta que pueda volver a actuar.

De esta forma, se crea una situación de urgencia ante estos momentos. Que los jugadores sepan que todo esto puede ocurrir en cualquier momento añade tensión al juego y obliga a los jugadores a que estén atentos.

Para evitar alargar el combate demasiado, si los enemigos controlados por el GM quedan fuera de combate pero no mueren, el mismo GM puede considerar que han muerto. Si el enemigo que queda fuera de combate es importante para la campaña, esta consideración puede evitarse.

5.2.10.4. Curaciones en combate

Todo daño infligido puede curarse siempre y cuando el personaje siga vivo. Para que un personaje deje de estar fuera de combate, necesita tener los PC y PA en positivo, por ende, necesita recuperar esos puntos perdidos, lo que simboliza curarse. Esto permite a los jugadores más control de la situación en medio del combate, y facilita también el propio combate.

Las curaciones no pueden exceder los PC o PA máximos del personaje curado. Durante el combate hay tres formas por las que un personaje puede curarse:

- Por habilidad “Cura”: Si un personaje usa Cura sobre otro personaje fuera de combate, este último recuperará los puntos que se hayan conseguido en la tirada. Si la curación logra poner en positivo los puntos del objetivo, este volverá a poder combatir en su turno.
- Por uso del kit médico: Un personaje puede usar un kit médico sobre otro personaje que esté fuera de combate para “estabilizarlo”, eso quiere decir que se recuperará con 1 punto en la estadística que tuviera en negativo.
- Por acción Medicina o Magicología: Si un personaje actúa con Medicina o Magicología con el afán de curar a un compañero que esté fuera de combate y logra superar la dificultad, conseguirá “estabilizarlo”, eso quiere decir que se recuperará con 1 PC o PA respectivamente.

5.2.11. Características del personaje

Hay ciertos elementos del personaje que, aunque no sean completamente necesarios para el correcto funcionamiento del juego, sí son útiles de tener determinados para poder lograr tener una mejor idea de su comportamiento y así, mejorar su interpretación. Esas son las características, información adicional del personaje que describe el comportamiento de un personaje. Esas características son:

- **Idiomas:** Simplemente los idiomas que puede hablar el personaje. En el mundo de Animaia las diferentes razas tienen formas únicas de comunicarse, con sus escrituras y vocabularios propios. Los idiomas que un personaje puede hablar deben tener cierta lógica dentro de su trasfondo. Los idiomas base del sistema de Animaia son:
    - Epiceno: Idioma con el que todas las razas se comunican entre ellas. Su lenguaje es una mezcla gramatical del resto de idiomas y su escritura se basa en símbolos formados por combinaciones de líneas, curvas y círculos. (Hablado como el inglés, escritura similar al coreano) Durante las partidas,

suele ser el que se usa de forma determinada para comunicarse.

- Humano: Idioma usado por la humanidad desde el principio de los tiempos y que ha ido evolucionando hasta ser lo que es en la actualidad. Se pueden estudiar el humano antiguo y el humano actual. La diferencia entre estos son que el idioma actual tiene una sintaxis optimizada del antiguo, el cual usaba más proposiciones y la gran mayoría de sus palabras contenían más letras. (Forma antigua: Hablado como el latín. Forma actual: Hablado como el español. Ambas escritas en alfabeto latino).
- Reptiliano: Idioma basado en la articulación de sus fonemas, suena duro a comparación al resto de idiomas debido a su frecuencia del uso de consonantes. (Hablado como el ruso, escritura similar al cirílico)
- Felino: Idioma complejo con muchas variantes el cual basa su sistema de escritura en el enlace entre letras de modo que cada letra puede tener hasta cuatro formas, según se escriba aislada, al principio, en medio o al final de la palabra. También tiene muchos dialectos diferentes. (Hablado y escrito como el árabe)
- Averio: Idioma muy particular basado en diferentes tonadas de silbidos. Para lograr comunicarse, combinan diversas frecuencias, longitudes y notas. Este idioma no tiene forma escrita.
- Mus: Idioma basado en dibujos que representan objetos físicos o elementos abstractos. Las uniones de diversos dibujos crean otras palabras con un significado vinculado a las palabras que lo forman. (Hablado y escrito como el chino)

- **Descripción:** Descripción sencilla, tanto física como emocional del personaje. Una explicación de la primera impresión que debería dar el personaje, sin entrar en muchos detalles.
- **Gustos:** Listado de las cosas que le gustan al personaje. Pueden listarse comida, sabores, colores, pasatiempos favoritos, ...
- **Aversiones:** Serie de cosas que no le gustan al personaje, como comida, tipos de gente, actividades, estilos de arte, ...
- **Ideales:** Explicación de las ideas que llevan al personaje a ser como es, como supersticiones, creencias o aspiraciones personales.
- **Vínculos:** Las relaciones personales o políticas que tiene el personaje con varios individuos o grupos.

5.2.12. Recursos y economía interna

Anteriormente ya se han nombrado todos los recursos con los que cuentan los personajes de los jugadores en este sistema.

Los Puntos de Cuerpo y Puntos de Alma son los recursos de combate, son recursos intangibles, la representación del estado corporal y el estado del alma.

El dinero almacenado en los monederos de los personajes son otro recurso, en este caso, el recurso económico, separado en 4 tipos de monedas, tangible para los personajes pero intangible para los jugadores.

Los objetos que guarden los personajes en su inventario son otro recurso, tangible para los personajes pero intangible para los jugadores.

En cada economía interna de un juego existen las siguientes mecánicas que hacen fluctuar los recursos de los personajes:

- Sources: Origen de donde se consigue recursos de forma automática. Son los elementos que hacen aparecer recursos “de la nada”.

- Drains: Elementos que consumen los recursos.
- Converters: Elementos que transforman recursos en otros recursos.
- Traders: Elementos que intercambian recursos por otros elementos o recursos.

Para cada recurso, estas son sus relaciones con cada mecánica comentada anteriormente. Puede que no estén todas, pero sí las más comunes:

	Sources	Drains	Converters	Traders
PC	Valores del personaje Habilidad Cura Descansar Uso de la acción "Medicina"	Ataques enemigos Habilidades enemigas	Uso del kit médico	
PA	Valores del personaje Habilidad Cura Habilidad Desalmar Descansar Uso de la acción "Magicología"	Uso de habilidades Habilidad Desalmar	Uso del kit médico	
Dinero	Uso de la acción "Robar" Recompensa de misiones Donaciones	Ser robado Pago por servicios Donaciones		Compraventa de objetos

Objetos	Encontrar Uso de la acción "Robar"	Ser robado Intercambio por servicios Uso del objeto Pérdida del objeto	Manipulación del objeto	Compraventa de objetos

5.3. Interfaces

Para poder hablar de las interfaces del proyecto, hay que dirigirse al diseño de la aplicación móvil que funciona en base al sistema de rol Animaia. Esta aplicación se usaría para la creación y gestión de personajes del sistema, por lo tanto debe permitir el uso de las mecánicas del sistema.

Se ha decidido que, para ese cometido, la aplicación se dividiera en 3 menús principales, el menú inicial, donde poder iniciar la creación de personajes o entrar al gestor de personaje; el menú del creador de personajes en sí, y el menú del gestor de personajes. Ver **Figura 5.3/1.**

Toda la aplicación comparte varios elementos en cuanto a diseño. Sus elementos recurrentes son el uso de botones táctiles distribuidos por la pantalla, la aparición de menús estilo pop-up para mostrar información o herramientas más concretas y el uso de fondos basados en pinturas acrílicas, simulando las texturas que frecuentemente tienen los dados que se suelen usar para juegos de rol, dados acrílicos. Varias de esas pinturas acrílicas son imágenes de uso libre creadas por NassyArt, mientras que otras son imágenes de stock. A parte de los elementos diseñados a base de imágenes reales o pinturas de acrílicos, hay elementos que usan colores básicos como base de su diseño. Estos son los bloques de información y herramientas del gestor de personajes, hechos así para diferenciarse de forma simple entre sí.

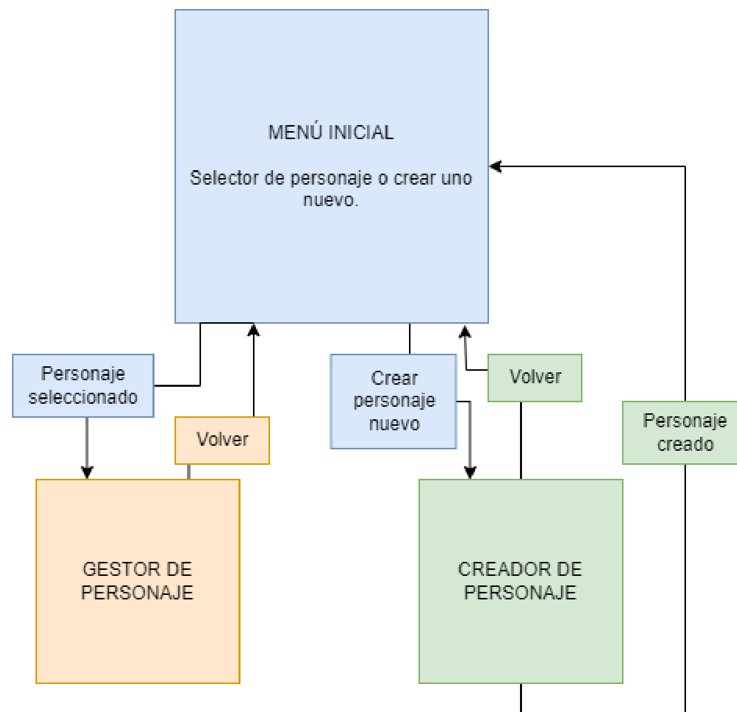

Figura 5.3/1: Diagrama de flujo de los menús de la aplicación. Cada dolor representa un menú. El origen de la acción de un color pertenece al menú del mismo color.

Al ser una aplicación de móvil, donde su funcionalidad es simplemente dar herramientas y mostrar la información necesaria a los jugadores para el correcto funcionamiento de las partidas de rol, las interfaces se consideran como no diegéticas.

En toda la aplicación se usa, al igual que en este documento, la fuente de texto Nunito. Ver **Figura 5.3/2**.

Penultimate

The spirit is willing but the flesh is weak

SCHADENFREUDE

3964 Elm Street and 1370 Rt. 21

The left hand does not know what the right hand is doing.

Figura 5.3/2: Imagen con una muestra de la fuente usada, Nunito.

En cuanto a los colores usados, mayoritariamente se han usado el color naranja para representar elementos vinculados con los valores corporales de los personajes y el color cyan para los elementos vinculados a los valores almáticos. La mayoría de menús pop-up tienen una imagen de colores verdosos como fondo, los botones de confirmación de esos menús tienen de fondo otra imágen de colores lilas. Otros botones usan imágenes de colores muy oscuros como fondo.

5.3.1. Menú Principal

Figura 5.3.1/1: Imágen del menú inicial en el inspector de Unity.

Este menú principal se puede dividir en dos partes, la superior, reservada para mostrar el icono del sistema de rol, y la parte inferior, donde se muestran los botones de selección o creación de personaje. Ver **Figura 5.3.1/1**.

Normalmente, para un jugador nuevo, solo aparecerán botones para crear un personaje, ya que aún no tiene uno. Esos botones seleccionados para crear un personaje se sustituyen por los botones de selección de personaje cuando ese personaje se haya creado.

El fondo de la parte superior del menú es una pintura acrílica de tonos azulados, mientras que el fondo inferior es la imágen de una mesa de madera, retocada usando el programa Adobe Photoshop. Esta última simboliza la mesa sobre la que se va a jugar la partida de rol.

El botón superior usa de fondo la imágen de un papel, representando la ficha de personaje, que suele ser en papel, también usado como fondo de otros botones y de fondo principal del menú del gestor de personaje. Ese tipo de botón muestra la información más destacable del personaje como su nombre, su raza y clase y finalmente, su nivel actual. También puede observarse un icono de una papelera, que sirve como botón para poder eliminar el personaje.

Los otros botones son los que llevan a la creación de personaje. Su fondo es una pintura acrílica de colores negros, usada en esta aplicación en muchos otros botones. Ver **Figura 5.3.1/2**.

Figura 5.3.1/2: Imágen del menú inicial en el inspector de Unity con la opción de borrado de personaje activada.

Cuando se quiere borrar un personaje, primero sale un aviso de confirmación, para evitar eliminar por error al personaje. Este aviso ocupa el espacio del botón del personaje, oscurece su fondo y muestra 2 botones grandes para cancelar o confirmar el borrado. Los colores de cada botón corresponden al rojo y verde respectivamente, colores que se asocian popularmente a cada acción, ya que el verde suele usarse como indicador de aprobación o acierto y el rojo para lo contrario, representando negación o error.

5.3.2. Creador de personajes

Figura 5.3.2/1: Imágen de la primera pantalla del creador de personajes.

Los elementos de diseño que comparte todo el creador de personajes son la barra superior de progreso, que sirve para mostrar al usuario el progreso de la creación del personaje para que entienda rápidamente cuántos pasos hay y cuánto falta en cada momento, el fondo, igual al del menú principal haciendo el paralelismo con la creación de un personaje en un sistema de rol clásico, escribiendo en la ficha de personaje sobre la mesa, y los botones inferiores, que sirven para avanzar y retroceder en el proceso de creación y que usan de fondo la misma imágen de los botones de creación de personaje. Además, todo el texto está en un gris muy claro para contrastar con el fondo.

En cuanto a elementos concretos, cada paso de este proceso puede dividirse en dos fases. La fase de descripción, donde se muestra información importante para entender cada paso, repite la misma estructura en la **Figura 5.3.2/1**, con el texto de la explicación del paso y la información importante al jugador. Se usa un título para cada paso para indicar el paso actual y un tamaño de texto grande en la explicación para que sea fácil de leer.

La segunda fase de cada paso consiste en una pantalla donde se da a elegir al jugador un número de opciones dependiendo del paso, para poder asignar cada elemento a su personaje. Estas pantallas vienen precedidas en la primera fase de una explicación de lo que deberán hacer los usuarios para que se entienda perfectamente cómo proceder en este tipo de pantallas. Para asegurar que la información se asigna correctamente, el botón para continuar al siguiente paso no estará disponible hasta que se haga la elección. Los botones usan la misma imágen vista anteriormente. Cuando se da a uno de esos botones suele salir un menú estilo pop-up con información extra de la opción seleccionada y otro botón que sirve para confirmar la elección. Siempre hay la opción de cerrar el menú y elegir otra alternativa.

En el paso de asignación de valores, antes de asignar todos los valores, se pregunta si se quiere confirmar la distribución de valores o reiniciarla, lo que volverá a iniciar el proceso con los valores iniciales (**Figura 5.3.2/3**). Esto permite infinitas oportunidades al usuario para poder distribuir los valores a placer. Siempre hay opción de reasignación de valores por si el jugador prefiere optimizar y re-ordenar los valores del personaje una vez ve los otros pasos y cuales valores se asignan con cada raza y clase. Ver figuras de la **5.3.2/2** a la **5.3.2/12.**

Figura 5.3.2/2: A la izquierda, imágen del paso de asignación de valores del personaje. A la derecha, imágen del menú pop-up al seleccionar un número.

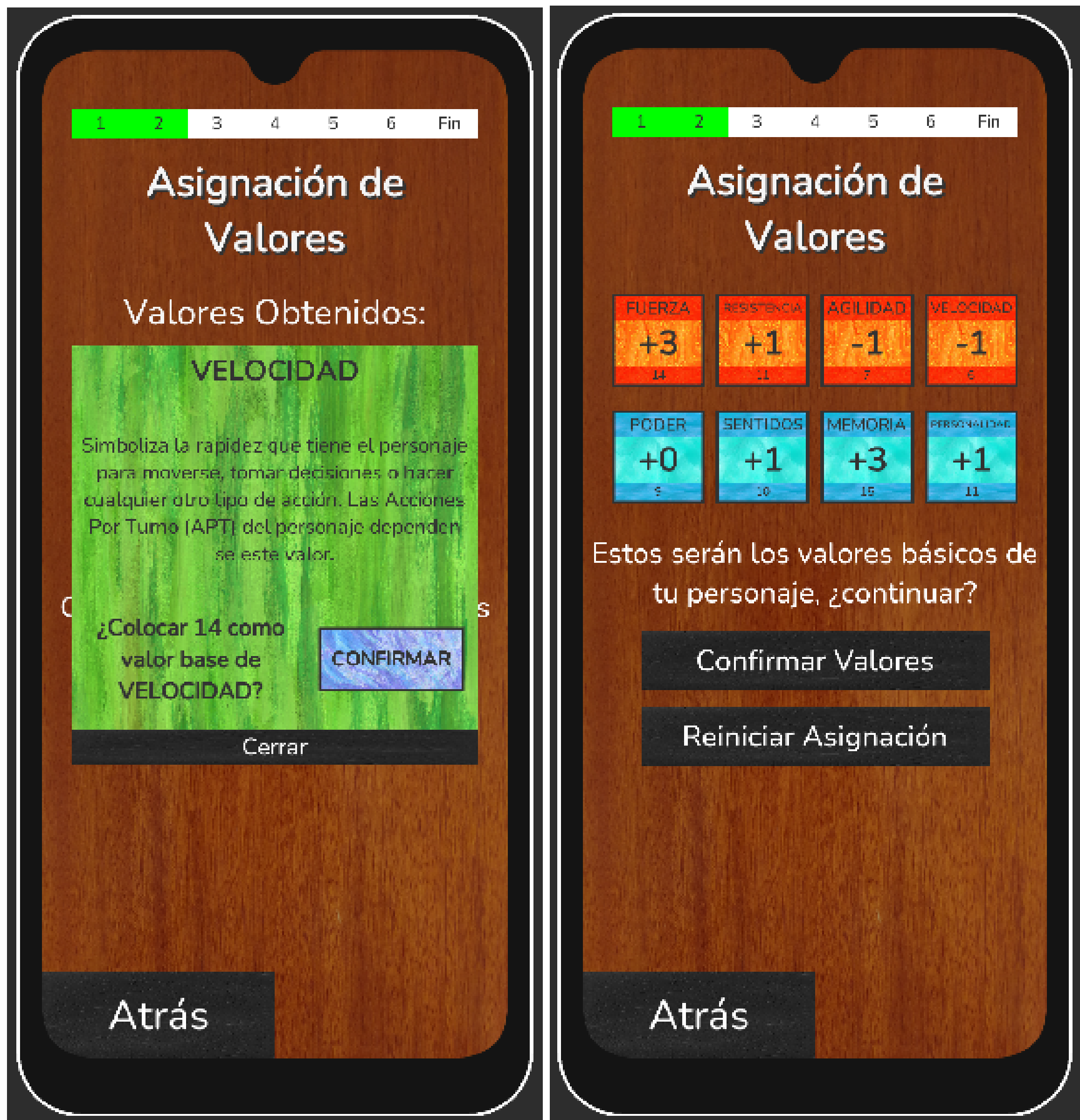

Figura 5.3.2/3: A la izquierda, segundo menú que aparece al elegir uno de los valores. A la derecha, imágen de la pregunta de confirmación de asignación de valores.

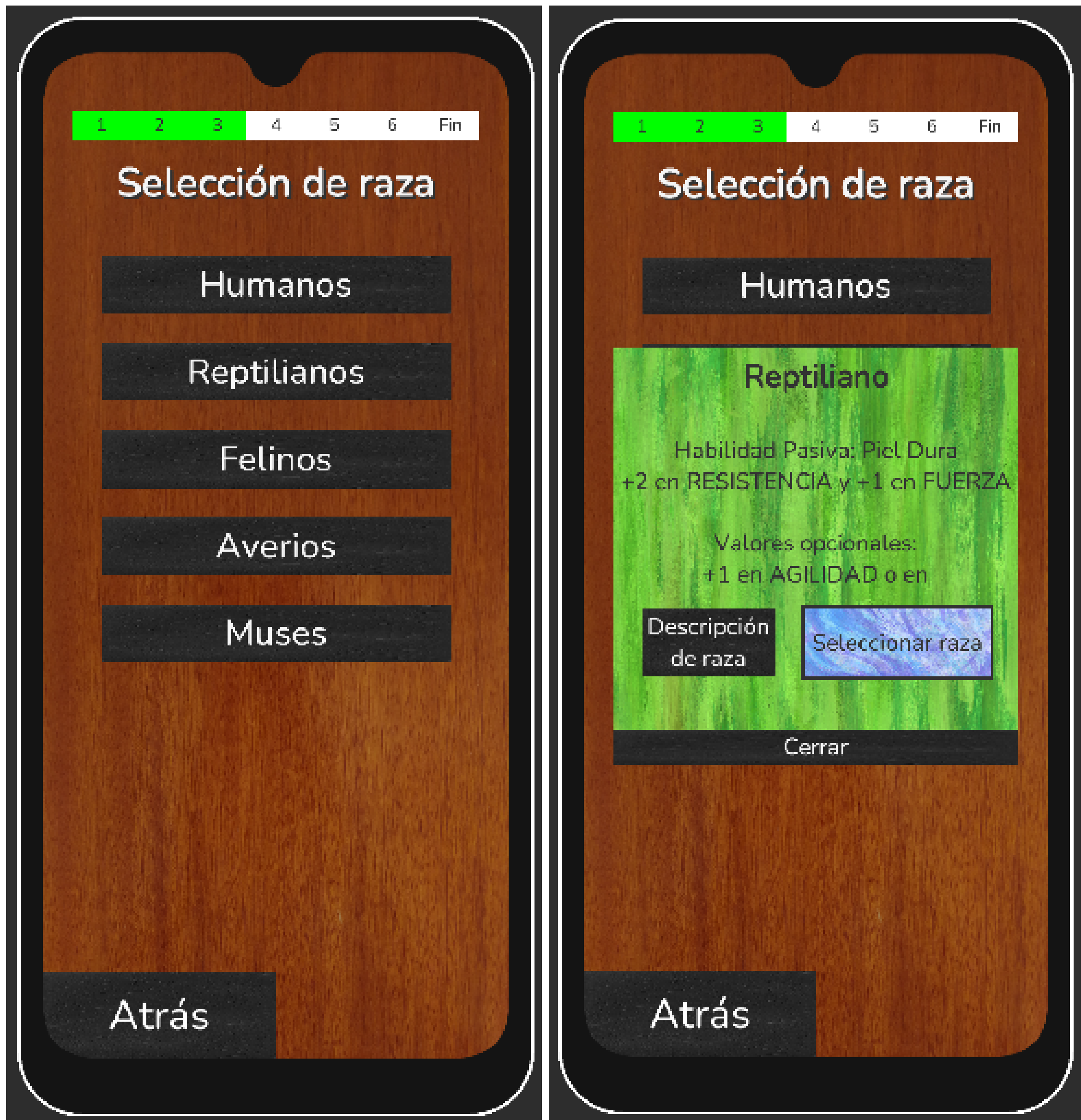

Figura 5.3.2/4: A la izquierda, imágen del paso de selección de raza. A la derecha, imágen del menú pop-up al seleccionar una raza.

Figura 5.3.2/5: A la izquierda, imágen del paso de selección del valor terciario de una raza, en este caso, Reptiliano. A la derecha, imágen de la pantalla al finalizar la selección de raza.

Figura 5.3.2/6: A la izquierda, imágen del menú principal pop-up del paso de selección de los valores de raza en caso de seleccionar la raza “Humanos”. A la derecha, imágen del mismo menú al pulsar el botón “VALORES CORPORALES”.

Figura 5.3.2/7: A la izquierda, imágen del menú principal pop-up del paso de selección de los valores de la raza “Humanos” al pulsar el botón “VALORES ALMÁTICOS”. A la derecha, imágen del paso de selección de clase.

Figura 5.3.2/8: A la izquierda, imágen del menú pop-up al seleccionar una clase. A la derecha, imágen de la pantalla al finalizar la selección de clase.

Figura 5.3.2/9: A la izquierda, imágen del paso de selección de arma. A la derecha, imágen del menú pop-up al seleccionar un arma.

Figura 5.3.2/10: A la izquierda, imágen de la pantalla al finalizar la selección del arma. A la derecha, imágen del paso de selección de las acciones ventaja.

Figura 5.3.2/11: A la izquierda, menú pop-up al seleccionar una acción. A la derecha, imágen de la pregunta de confirmación de la asignación de acciones ventaja.

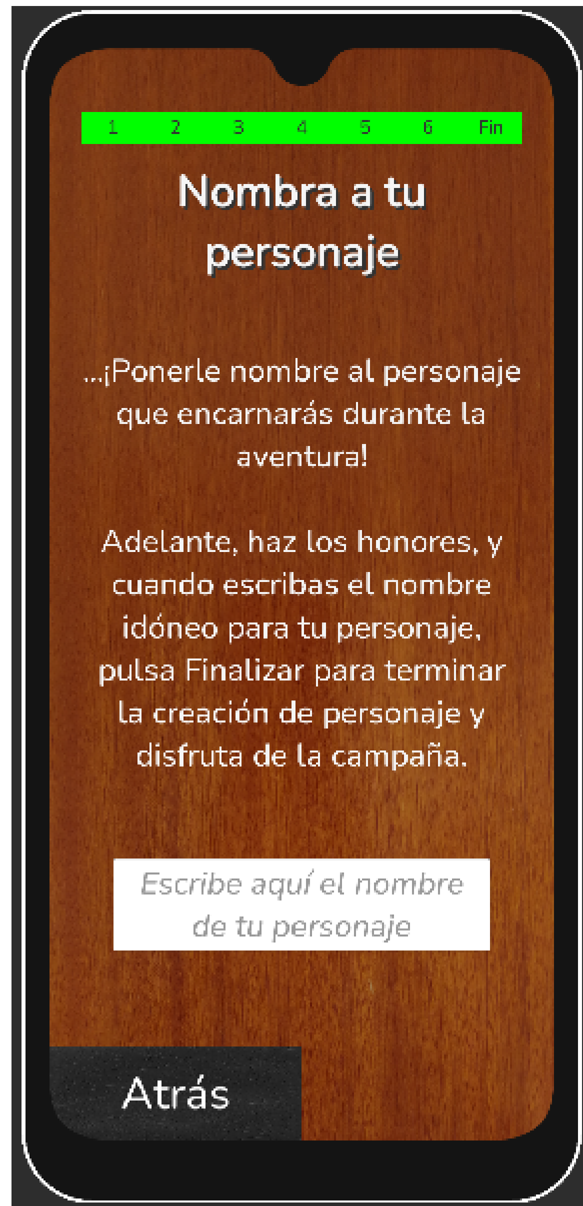

Figura 5.3.2/12: Imágen del último paso de creación para nombrar el personaje.

5.3.3. Gestor de personaje

El gestor de personajes de la aplicación está dividido en dos partes fundamentales. La parte superior, que es donde se muestra la información del personaje que también se muestra en el momento de seleccionarlo en el menú inicial, nombre, raza, clase y nivel. Esta parte es visible siempre.

En la parte inferior, que ocupa la mayor parte de la pantalla, está pensada para visualizar los diferentes bloques de información del personaje según su tipo. Estos bloques se visualizan mediante el uso de un botón que divide las dos

partes. El botón, que es desplegable, muestra la selección de todos los bloques de información disponibles, y al pulsar uno de ellos cambia la visualización al bloque seleccionado. Estos bloques son:

- **Valores:** Muestra un listado de los valores del personaje y funciones que influyen dichos valores. **Ver Figura 5.3.3/1**.
- **Acciones:** Muestra un listado de las acciones disponibles, tanto para la interpretación como para el combate. Ver **Figura 5.3.3/6.**
- **Habilidades:** Muestra el listado de habilidades del personaje. Ver **Figura 5.3.3/9**.
- **Equipo:** Muestra información del arma equipada, el inventario y el monedero del personaje, con herramientas para poder manipularlos. Ver **Figura 5.3.3/12**.
- **Características:** Muestra el listado de características y permite rellenar esos parámetros libremente. Ver **Figura 5.3.3/13**.
- **Manual del sistema:** Muestra el manual del sistema íntegro en caso de que el jugador necesite resolver alguna duda sobre el funcionamiento del juego. Ver **Figura 5.3.3/14**.

Cada herramienta que se ofrece para automatizar mecánicas del sistema funciona a través del uso de botones y menús estilo pop-up como los vistos en la creación de personaje, para que el jugador ya esté acostumbrado a su funcionamiento si ha navegado por el creador. Ver figuras de la **5.3.3/2** a la **5.3.3/14**.

Figura 5.3.3/1: Imágen del bloque “Valores” del gestor de personajes.

Figura 5.3.3/2: A la izquierda, imágen del menú pop-up de los datos generales del personaje. A la derecha, imágen del menú pop-up de la experiencia del personaje.

Figura 5.3.3/3: A la izquierda, imágen del menú pop-up del primer paso de la subida de nivel del personaje. A la derecha, imágen del mismo menú en el segundo paso de la subida de nivel del personaje, similar al visto en la figura 5.3.2/6.

Figura 5.3.3/4: A la izquierda, imágen del menú pop-up de los valores corporales. A la derecha, imágen del menú pop-up de los valores almáticos.

Figura 5.3.3/5: A la izquierda, imágen del menú pop-up del valor “Cuerpo”. A la derecha, imágen del menú pop-up del valor “Alma”.

Figura 5.3.3/6: A la izquierda, imágen del menú pop-up de la acción “Descansar”. A la derecha, imágen del bloque “Acciones” del gestor de personajes.

Figura 5.3.3/7: A la izquierda, imágen del menú pop-up del uso de acciones predeterminadas. A la derecha, imágen del menú pop-up para atacar con el arma equipada.

Figura 5.3.3/8: A la izquierda, imágen del menú pop-up del resultado del ataque con arma. A la derecha, imágen del menú pop-up del resultado de la esquiva/bloqueo del ataque.

Figura 5.3.3/9: A la izquierda, imágen del menú pop-up del resultado de esquiva/bloqueo al ser atacado. A la derecha, imágen del bloque “Habilidades” del gestor de personajes con huecos de habilidad bloqueados y disponibles.

Figura 5.3.3/10: A la izquierda, imágen del menú pop-up para añadir una nueva habilidad. A la derecha, imágen del menú pop-up para elegir la habilidad seleccionada en el menú anterior.

Figura 5.3.3/11: A la izquierda, imágen del menú pop-up para usar una habilidad. A la derecha, imágen del menú pop-up del resultado de la acción. También es visible el diseño de las habilidades disponibles en el bloque de "Habilidades".

Figura 5.3.3/12: A la izquierda, imágen del bloque "Equipo". A la derecha, imágen del menú pop-up para manipular la cantidad de la moneda seleccionada.

Figura 5.3.3/13: A la izquierda, imágen del menú pop-up de información adicional del equipo, en este caso, del Monedero. A la derecha, imágen del bloque “Características”.

Figura 5.3.3/14: A la izquierda, imágen del bloque “Manual del sistema”. A la derecha, imágen del botón desplegable de navegación con las opciones visibles.

5.4. Narrativa

Este juego de rol se basa en la creación de una historia a partir de las acciones tomadas por los jugadores. Es por eso que no existe una historia concreta, el objetivo real de este juego es hacer que el GM junto con los jugadores puedan crear una trepidante historia sobre cómo sus personajes consiguieron realizar increíbles hazañas juntos y crear momentos memorables para todos los participantes. El sistema de rol Animaia sólo debe ofrecer el trasfondo base del mundo en el que ocurren todas las historias.

5.4.1. Sinopsis

En el sistema de Animaia, el juego no tiene ninguna sinopsis determinada y las historias pueden ser de cualquier cosa que quiera el GM, quien tiene completa libertad creativa para inventar la historia que desee o inventarla según la marcha de las mismas partidas a partir de las consecuencias generadas por las acciones de los personajes. A dicha historia se le puede llamar "campaña". Las campañas pueden no tener una longitud establecida, pudiendo continuar durante meses o incluso años o pueden ser "one-shots", campañas pensadas para jugarse en una sola partida.

5.4.2. Trasfondo del mundo

El mundo donde ocurren todas las historias de este juego de rol se llama Animaia, un planeta donde fluye una energía que concede a los seres que viven en ella en mayor o menor medida, poderes sobrenaturales, conocida como alma, energía almática o poder almático.

Todos los seres de este mundo tienen un alma, concedida por el propio planeta en el momento de nacer. Esta alma es un cúmulo de energía que permite a los seres tener voluntad, y es donde se almacenan la personalidad y todos los recuerdos vividos por el ser. A medida que ese ser envejece, su alma aumenta para poder almacenar todos esos recuerdos, y en el momento de perecer, su alma vuelve al planeta. Cuando eso ocurre, el planeta absorbe y se enriquece con toda la información almacenada en ese alma y aprende para poder mejorar su propio funcionamiento. Entonces el bucle se repite.

A cada ciclo, Animaia consigue conocimientos y poder para ser capaz de otorgar vida a más seres, cada vez más sofisticados. Su objetivo sigue siendo en la actualidad un completo misterio, aunque existen algunas teorías sobre el propósito de este ciclo.

La voluntad es lo que permite a un ser poder vivir y cumplir sus objetivos, es lo que les permite convertirse en versiones mejoradas de sí mismos. La voluntad la genera el propio ser a partir de su personalidad o ego.

Animaia reparte energía almática a los seres recién nacidos según la voluntad de sus antecesores para continuar la cadena evolutiva. Es por eso que, de cierta forma, todos los seres del planeta heredan la voluntad de sus antepasados. Cuanta mayor sea la voluntad, mayor será su alma y por lo tanto, más sofisticados serán esos seres.

Cuando un ser adquiere una voluntad muy grande, su alma crece simultáneamente, volviéndose un gran alma, y es entonces cuando llega al punto de evolución hasta tener raciocinio. Es por eso que en este mundo hay hasta 5 razas que han obtenido completa conciencia y raciocinio.

Cuando un ser pensante es consciente de su propia alma, es capaz de canalizar su propia energía de tal forma que puede ejecutar habilidades sobrenaturales, las cuales no podría hacer sin el uso de su propio poder almático.

Los jugadores pueden inventarse el trasfondo o los países y sus situaciones sociales y geopolíticas de las razas existentes en este sistema para poder situar sus historias de la forma que mejor les convenga. Igualmente, en este manual se ofrecen unos trasfondos predeterminados de las razas disponibles.

Como ejemplo, en la actualidad, las 5 diferentes razas han logrado crear civilizaciones que después de varios siglos de grandes conflictos entre ellas, han conseguido una relativa paz, con una repartición de territorios más o menos equitativa. En muchas partes del mundo, todas las razas viven entremezcladas en núcleos urbanos, aunque sigan existiendo diversos conflictos debido a las diferencias de ideales y perspectivas de la vida. Hay países donde todas las razas viven

juntas mientras que hay otros en los que solo una raza vive y no deja entrar al resto de razas o se repudia una raza en concreto.

5.4.3. Narrative devices

El GM es quien dirige la historia y hace de narrador de la misma, explicando las consecuencias de cada acción e interpretando a los personajes que no controlen el resto de jugadores. Prepara la historia y las situaciones en las que se encontrarán los jugadores durante las partidas y si es necesario, puede improvisar y cambiar el desarrollo de esta.

5.4.4. Tensión dramática y jugable

Toda la tensión del juego se origina en la aleatoriedad de todas las consecuencias provocadas por las tiradas de dado, además de las dificultades de las acciones y situaciones establecidas por el GM. Que los jugadores no sepan cómo resultarán las acciones que ejecutan es la fuente principal de tensión de este juego.

5.4.5. Puntos de la trama, granularidad y disparadores

Todos estos elementos de la narrativa en este sistema, dependerán de cómo el GM quiera estructurar la trama de las partidas y de las acciones de los jugadores y las consecuencias que éstas acarrean.

Al ser un juego muy situacional y no tener una historia concreta, todos estos elementos varían de partida en partida y de campaña en campaña dependiendo de las preferencias de todos los participantes.

5.4.6. Estructura narrativa

Se recomienda dividir las campañas en misiones, siguiendo una estructura episódica. Cada misión suele estar pensada para

poder jugarse en pocas partidas, ofreciendo retos planificados para poder superarse con trabajo en equipo. De esta forma, los jugadores siempre tienen un claro objetivo a cumplir. El GM debe tener en cuenta muchas posibles consecuencias para que la historia siga un rumbo lógico. Por lo tanto, hay que vaticinar dichas posibles consecuencias en la medida de lo posible. Aunque esa sea la estructura más recomendable, el GM es libre de decidir cuál estructura quiere que siga la campaña.

5.4.7. Dimensión física

En este sistema, todas las partidas ocurren en el mundo ficticio de Animaia. Este mundo sigue todas las leyes físicas del mundo real, a excepción de su inherente poder almático. Sus localizaciones son muy similares a las del mundo real.

Una historia puede abarcar la aventura dentro de una sola ciudad, de un país o incluso un continente. Eso depende de la historia que se quiera seguir por parte del GM y los jugadores.

5.4.7. Dimensión temporal

El paso del tiempo en Animaia es el mismo que en el mundo real, excepto en algunas situaciones. Por ejemplo, ejecutar una acción como buscar algo en una habitación puede llevar varios minutos, mientras que intentar intimidar a alguien solo lleva unos pocos segundos. Viajar de una villa a otra puede tomar horas si se va a pié, mientras que si se hace el mismo recorrido en montura, el tiempo se reduce adecuadamente al tipo de montura. El tiempo transcurrido entonces, depende de las acciones tomadas y de la situación. Esto debe establecerlo el GM, tomando en cuenta los aspectos mencionados.

Durante el combate, cada acción tomada solo tarda unos pocos segundos, ya que suele ser una situación frenética. Las habilidades se ejecutan casi instantáneamente, y los ataques también son muy rápidos, haciendo que los turnos de un

personaje representen solo una docena de segundos como mucho.

5.4.8. Dimensión ambiental y referentes estéticos

El mundo de Animaia se basa en un mundo de magia y fantasía, basado en la época de los inicios de la Edad Moderna. Aparte de este contexto, el resto de detalles ambientales como los estilos arquitectónicos, climatologías, estilos de vestimenta y otros son decisión del GM.

No concretar estos detalles permite al GM ambientar la historia de la campaña con total libertad, posibilitando la creación de mundos únicos dentro del mismo sistema. La comunidad puede de esta forma crear un multiverso basado en Animaia, igual que se ha hecho con muchos otros sistemas de rol. Además, todos los jugadores pueden tomar los referentes estéticos que quieran en el momento de idear las campañas o crear sus personajes.

5.4.9. Dimensión emocional

Al igual que muchos de los aspectos comentados, está en manos del GM decidir cuales son las emociones que quiere transmitir en momentos concretos. Esto puede depender de las situaciones en las que tienen que pasar los personajes, y puede llegar a ser muy variable debido a la aleatoriedad intrínseca de los desenlaces que pueden llegar a ocurrir en cualquier momento.

5.4.10. Aspectos éticos

Al tener el GM la total libertad de decisión de la historia, es el que decide cuando y como los personajes protagonistas deben atravesar situaciones en las que su moralidad es cuestionada,

desde cualquier punto de vista. Estas situaciones pueden ser muy útiles para desarrollar la trama y los personajes.

Por otra parte, son los jugadores quienes deciden la moralidad de sus personajes, son completamente libres de actuar como quieran ante las situaciones que el GM les plantea.

5.5. Personajes

Son el elemento principal de cualquier sistema de rol, su mecánica base. Es a través de estos que los jugadores actúan durante las partidas, son sus avatares dentro del mundo de juego. Cada personaje controlado por un jugador es el protagonista de la historia que se desarrollará según sus decisiones a lo largo de las partidas. El desarrollo de personaje depende del jugador que lo controla.

Todos los personajes tienen ciertos atributos que permiten su interpretación de forma funcional durante las partidas, que son los valores, la raza, la clase, el equipo y las habilidades. Todos estos aspectos ya se han comentado en el apartado de mecánicas como funcionan y para qué sirven, así que no se volverán a comentar en detalle en este apartado.

Lo que sí se debe comentar son los elementos que ofrecen información a los jugadores sobre los comportamientos y características físicas de los personajes, que ayudan a entender cómo interpretar correctamente cada tipo de personaje.

Para empezar, este sistema da completa libertad a los jugadores a imaginar e interpretar a sus personajes como quieran, ya que dichos personajes no son más que sus propias creaciones. Son libres de presentar a sus personajes de la forma que les plazca. Es por esto que este sistema no tiene un diseño de personajes muy concreto en cuanto a los atributos físicos y psicológicos, eso queda en manos de los propios jugadores para ofrecerles total libertad creativa. Aún así, sí que existen algunos elementos que dan ciertas características a los personajes para que los jugadores puedan hacerse una idea simple, sin ser nada concreto, de cómo son los personajes que interpretan. Estas son la raza y clase a la que pertenecen.

5.5.1. Razas

Cada raza describe las características físicas generales de los personajes, y añade un trasfondo simple en cuanto el comportamientos de estos personajes hacia personajes de otras razas. Sirven como una guía que genera la idea principal de cómo es el personaje que el jugador está creando, una plantilla muy simple. Los jugadores deben imaginar el resto de características a partir de la información que ofrece la descripción de cada raza. A continuación se nombrarán las descripciones de cada raza:

- **Humanos:** Los seres más comunes del planeta, y los más versátiles. Pueden hacer de todo si se lo proponen. Se dice que eso es posible debido a que la voluntad de la humanidad es infinita. Están esparcidos por todos los rincones del mundo y gobiernan la gran mayoría de la población. Su control sobre todas las sociedades es casi absoluto gracias a la capacidad que tienen para cooperar y negociar con el resto de razas a lo largo de la historia. Cuando la política no daba sus frutos, su gran capacidad de estratagemas en combate y sus avances en casi cada campo científico les aseguraba casi siempre la ventaja, así como su dominio en el campo de batalla. Se relacionan generalmente bien con todas las razas, viviendo en armonía en los centros urbanos, pero hay pequeños grupos que aún siguen teniendo rencor a otras razas debido a las diferencias de cultura y guerras del pasado.

- **Reptilianos:** La forma perfeccionada de los reptiles. Seres de gran dureza, longevidad y fuerza gracias a su piel escamada además de una gran ambición. Debido a su portento físico, suelen dedicarse a trabajos donde la fuerza reina, y sus sociedades suelen regirse por la ley del más fuerte. Muchos de ellos prefieren vivir en grupos exclusivos de reptilianos, pero hay algunos que prefieren alejarse de los nidos y mezclarse con el resto de culturas. Antiguamente solían mantener muchas guerras por control territorial contra el resto de razas. Debido a la multitud de guerras perdidas contra los humanos, suelen

tener rencor hacia ellos, y mantienen una gran rivalidad hacia los felinos. No tienen grandes opiniones sobre los averios y muses, con los que a veces hasta han cooperado en el pasado. Se dice que hay un gran grupo de reptilianos que pretenden hacerse con el control mundial desde las sombras, aunque esto solo sean rumores con apenas fundamentos.

- **Felinos:** Seres de gran agilidad, con rasgos parecidos a los gatos. Son conocidos por sus gráciles movimientos, sigilo y su enorme ego. Su forma de moverse les permite tener la ventaja a la hora de emboscar a sus enemigos o de evitar sus ataques. Los hay que prefieren vivir en manada cooperando entre ellos defendiendo su territorio, otros que prefieren buscar la compañía de alguien de diferente raza y otros que prefieren hacer las cosas en completa soledad. Su rivalidad con los reptilianos la consideran como una rivalidad infantil y una guerra de egos. En realidad su mayor rivalidad es hacia los averios seguida de la que tienen con los muses, quienes han estado cazando durante toda la historia por considerarlos como nada más que alimento. A los únicos que no ven como seres inferiores y que incluso tratan como iguales son los humanos, quienes han ayudado desde tiempos inmemoriales. Se dice que su alianza con los humanos es la más antigua de la historia.

- **Averios:** Seres con la capacidad de volar y rasgos de pájaro. Su capacidad de volar les hace únicos. Destacan por su sensibilidad en varios sentidos como el oído o la vista. Eso les permite ser increíbles cazadores. Algunos grupos de averios guardan cierta rivalidad con los muses ya que los consideran como su principal alimento aunque la mayoría tengan buena relación con ellos, hasta el punto de tener cierta amistad y alianzas para defenderse de los felinos. Esto también los ha llevado a tener ciertos vínculos de amistad con algunos grupos de reptilianos

para repeler el acoso de los felinos. Igual que el resto de razas, los averios suelen ser amistosos con los humanos, quienes les han ayudado para reducir las hostilidades entre ellos y los felinos.

- **Muses:** Pequeños seres peludos similares a los roedores. Guardan un gran vínculo con el alma, y son los seres que mayor capacidad tienen de manipularla igualados solo por ciertos humanos. Junto con los averios, se han visto obligados a cooperar con los reptilianos para defenderse de los felinos, a quienes guardan un gran odio. Son los seres más comunes en las ciudades junto a los humanos aunque también hay grandes poblaciones de ellos en zonas rurales. Son seres muy sociables y prefieren hacerlo todo en grupo, ya sea entre muses o otras razas. A estos seres les fascina aprender sobre todos los misterios que guarda la misma realidad, por eso cooperan de tú a tú con los humanos sobre todo en la gran mayoría de campos de la ciencia y la tecnología. Se dice que es gracias a ellos que los humanos han podido prosperar y desarrollar todas las invenciones que han creado, incluso se dice que la manipulación almática fue una enseñanza de los muses hacia los humanos.

A partir de estas descripciones, los jugadores deben imaginar a sus personajes, asociándolos a una raza en concreto de los animales en los que se basan estas razas y tomando sus características como las características físicas y psicológicas de sus personajes.

5.5.2. Clases

Las clases describen no solo la característica en la que destacará un personaje, sinó también su estilo de combate, el arma que va a usar y las habilidades que va a aprender. Cada clase ofrece un poco de información sobre el comportamiento de los personajes en batalla, aunque no son un indicativo que

obligan a los jugadores a mantener esos comportamientos para sus personajes. A continuación se mostrarán las descripciones de cada clase:

- **Aniquilador:** Viven por y para el combate cuerpo a cuerpo. Su extraordinaria fuerza y agresividad les permite destrozar a sus oponentes en devastadores ataques que diezman sus defensas. No hay protección posible ante un aniquilador.

- **Protector:** Gracias a su vitalidad y resistencia física los convierte en los personajes idóneos para hacer las funciones de "muro" en combate, absorbiendo todos los golpes de los enemigos mientras que sus compañeros terminan con ellos. Los hay que prefieren usar sus conocimientos de medicina, sus cuerpos entrenados y sus habilidades de manipular su propia alma para aplicar hechizos curativos en combate.

- **Cazador:** Cuando tienen un objetivo, éste sólo está condenado a ser capturado, ya sea un objeto, un ser vivo o un elemento metafórico. Hacen uso de su increíble agilidad y destreza con las manos para atacar y moverse entre las filas enemigas grácilmente. Todos sus movimientos son óptimos, y eso los hacen perfectos para el subterfugio y el sigilo.

- **Sombra:** Usan su extremadamente alta velocidad para arremeter contra el enemigo en un parpadeo sin posibilidad de respuesta o para someterlos con un torrente de ataques. Su rapidez deja a sus enemigos sin escapatoria.

- **Animatec:** Conocedores absolutos sobre la manipulación almática, ya sea por genética o riguroso estudio, han perfeccionado su uso a tal punto que pueden usar la forma más pura y eficiente de su alma para proyectar hechizos. Solo así se consigue proyectar los hechizos más poderosos conocidos actualmente.

- **Centinela:** Aclamados por sus capacidades a la hora de encontrar cualquier cosa gracias a sus agudizados

sentidos, suelen dedicarse a ser los vigías de cualquier grupo. Suelen adelantarse a los acontecimientos porque se han percatado mucho antes gracias a esos increíbles sentidos.

- **Sabio:** Años de estudios les han convertido en enciclopedias andantes de conocimientos. Puede que conozcan muchas cosas sobre una gran variedad de sujetos o que hayan dedicado sus vidas a desentrañar los misterios sobre algo en específico. No hay rama del conocimiento que les detenga para recopilar información e investigar sobre ellas.

- **Embaucador:** Temidos por su carisma y labia. Su capacidad de encandilar cualquier conversación hacen que sean capaces de conseguir lo que quieran con solo hablar. Los hay que han empezado y terminado guerras sólo con un discurso suyo.

5.6. Elementos de feedback

Para ofrecer una mejor experiencia de usuario, la aplicación debe responder a ciertas acciones para asegurar que entiende de forma clara las consecuencias de esas acciones. Es por eso que se han diseñado los siguientes elementos de feedback:

- **Sonido al hacer una tirada de dados:** Para que el jugador entienda que acaba de realizar una tirada de dados, se reproduce un audio de dados siendo lanzados sobre una mesa. Este elemento también añade inmersión a la experiencia de uso de la aplicación.

- **Desglose del resultado de una tirada:** A cada resultado, se muestra el desglose de la suma de todas las partes que conforman el cálculo total de ese resultado para que el usuario entienda su naturaleza por completo.

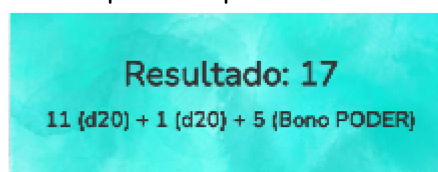

Figura 5.6/1: Desglose del uso de una habilidad.

5.7. Assets usados

En este apartado se listarán todos los elementos usados y creados para diseñar la aplicación móvil.

Figura 5.7/1: Logo principal del sistema de rol Animaia.

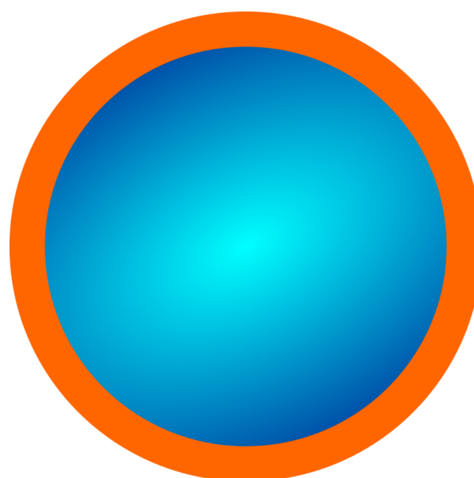

Figura 5.7/2: Icono de la aplicación móvil.

Figura 5.7/3: Icono de ayuda para el bloque “Equipo” del gestor de personajes (Negativo).

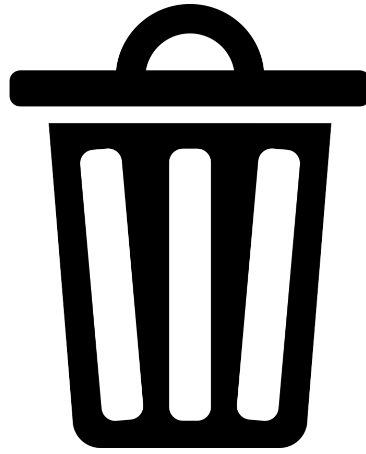

Figura 5.7/4: Icono de borrar para el borrado de personajes.

Figura 5.7/5: Imágen de un papel usada para fondos de menús y de botones.

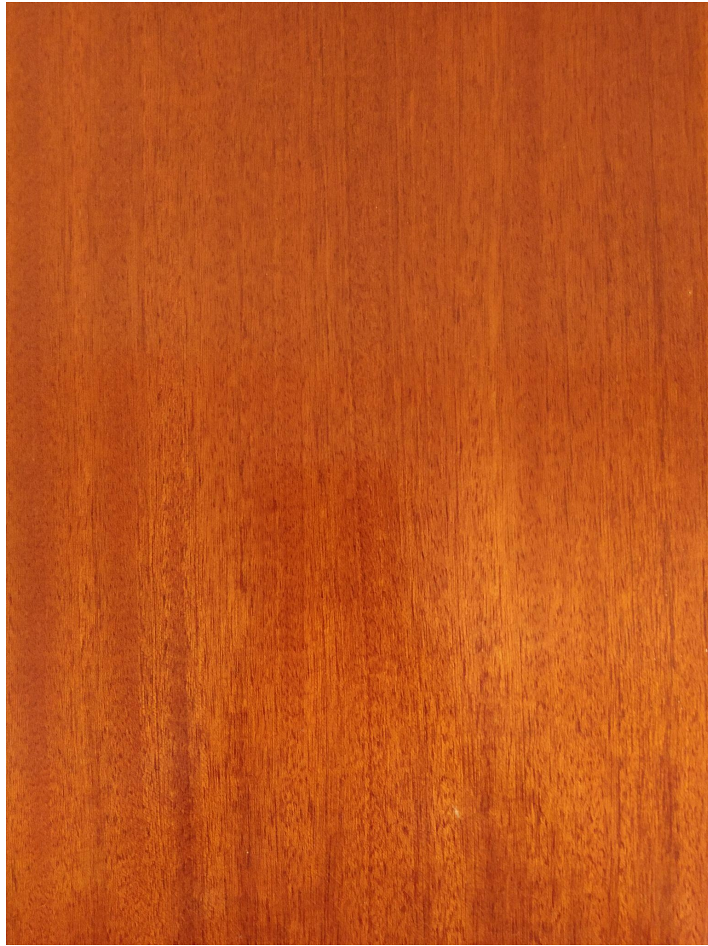

Figura 5.7/6: Imágen de una mesa de madera usada para fondos de los menús inicial y del creador de personajes.

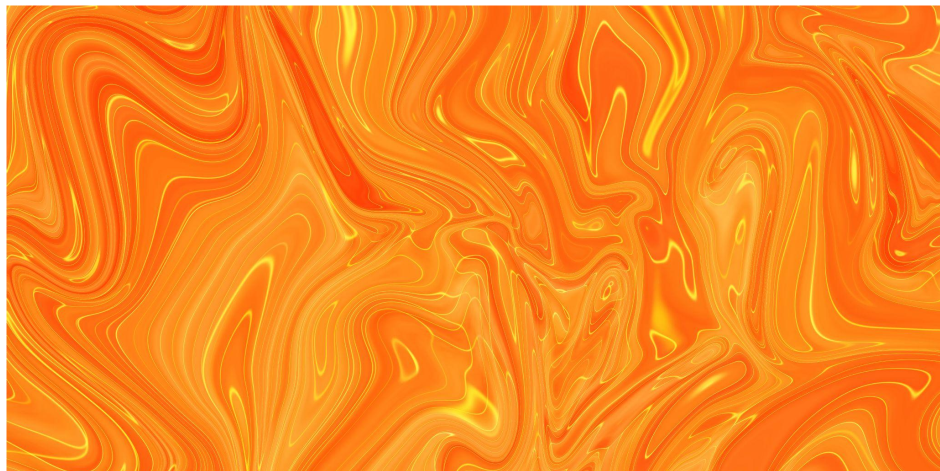

Figura 5.7/7: Imágen de una pintura acrílica usada para fondos de los menús y elementos relacionados con los valores físicos.

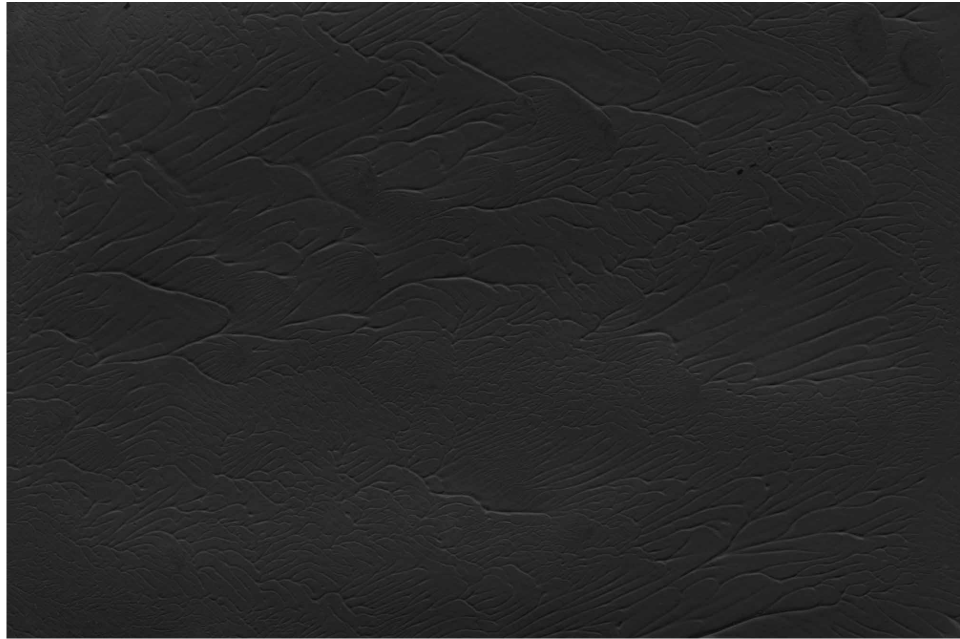

Figura 5.7/8: Imágen de una pintura acrílica usada para fondos de múltiples botones.

Figura 5.7/9: Imágen de una pintura acrílica usada para fondos de los menús y elementos relacionados con los valores almáticos.

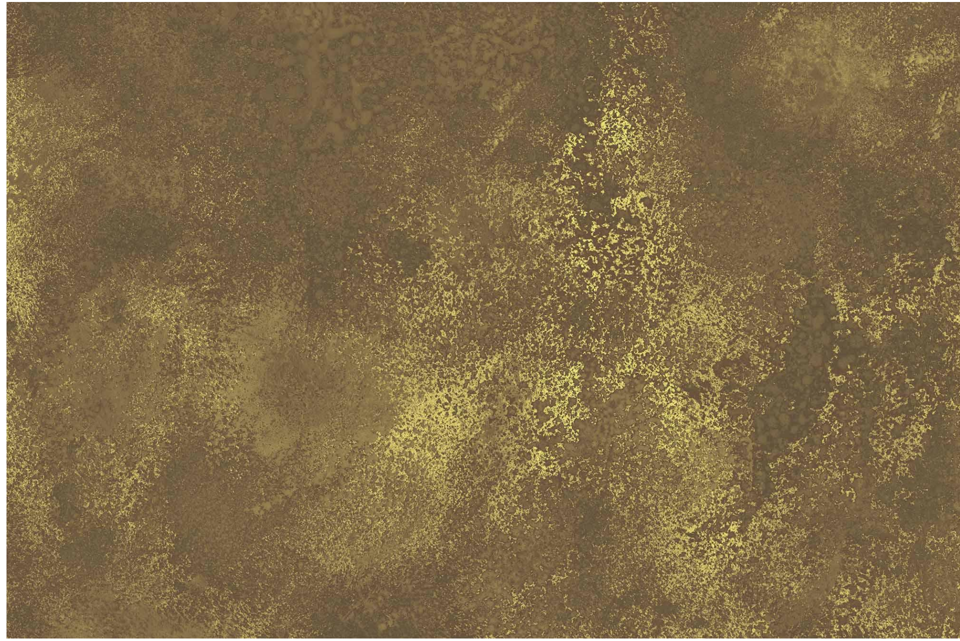

Figura 5.7/10: Imágen de una pintura acrílica usada para fondos de los valores al seleccionar el valor terciario de la raza.

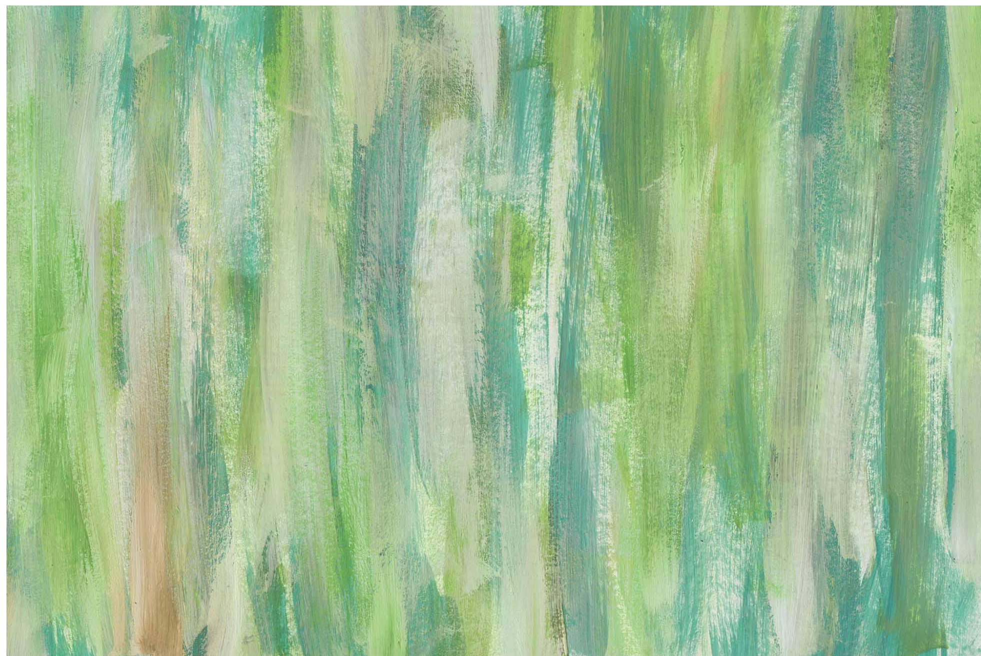

Figura 5.7/11: Imágen de una pintura acrílica usada para fondos varios menús.

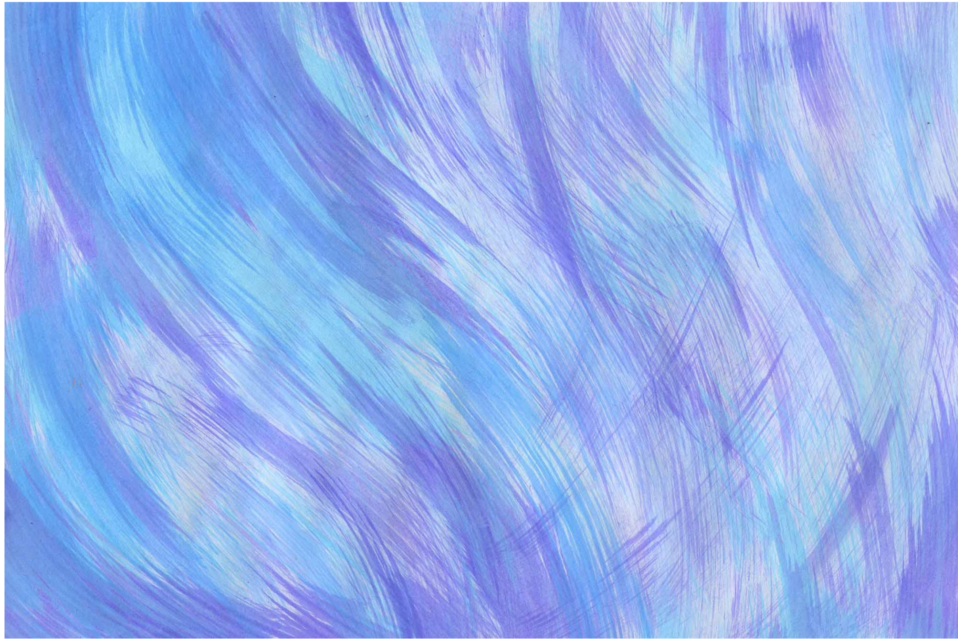

Figura 5.7/12: Imágen de una pintura acrílica usada para fondos de los botones de confirmación, ejecución de acciones y uso de habilidades.

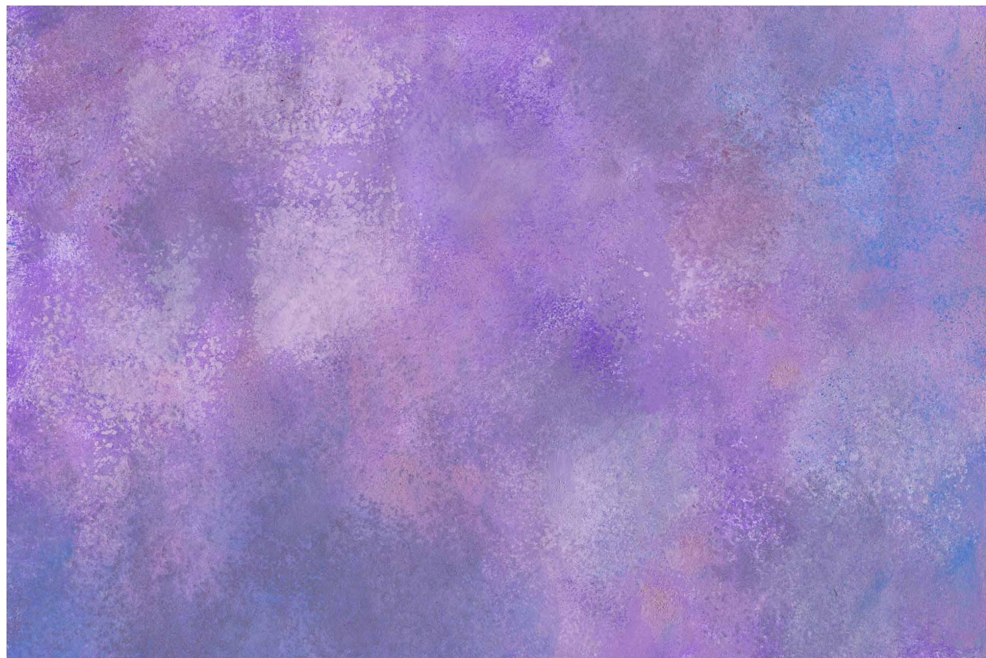

Figura 5.7/13: Imágen de una pintura acrílica usada para fondos de los valores al seleccionar cuál valor de raza mejorar al subir de nivel.

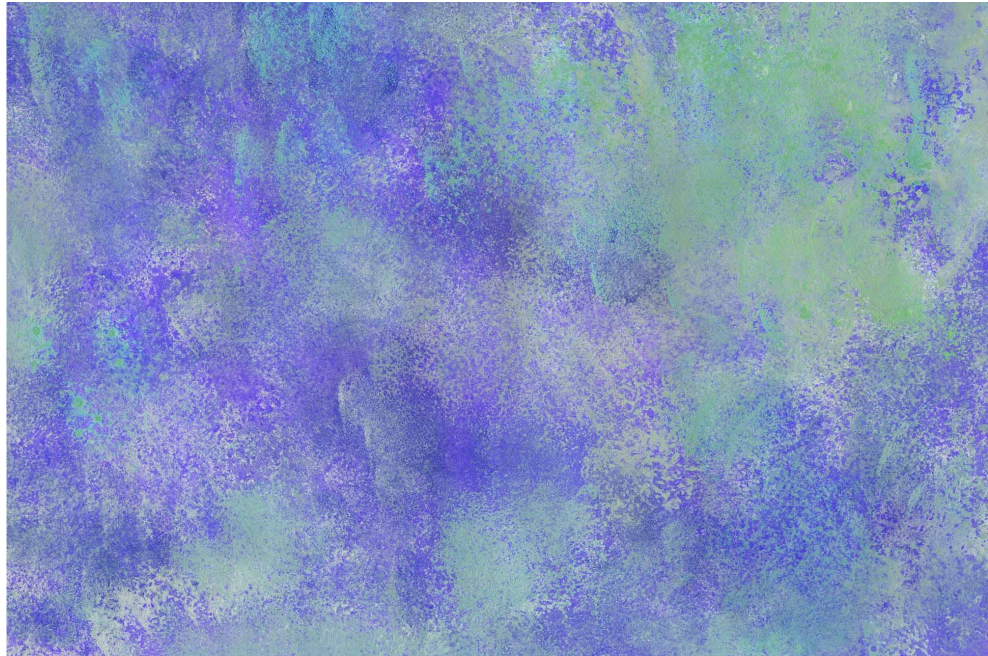

Figura 5.7/14: Imágen de una pintura acrílica usada en los fondos de la parte superior del menú inicial y del gestor de personajes.

5.8. Guardado y borrado de personajes

Para permitir el uso de múltiples personajes en la misma aplicación, y que se mantengan esos personajes entre partida y partida, con todos sus elementos guardados, se ha hecho uso del patrón de diseño *Singleton*.

Singleton es un patrón de diseño de software que permite el uso de una única instancia de una clase en toda la aplicación, en este caso, de la clase "Personaje". Esto permite seleccionar el personaje que se quiera y cargar sus respectivos datos de forma correcta entre las diferentes escenas. Esto es usado para poder crear el personaje en la escena del creador de personajes, asignarle todos los datos necesarios y que al terminar y volver al menú principal, la aplicación sepa que el personaje existe y lo muestre en el menú. Si se selecciona el botón de dicho personaje, entonces la aplicación sabrá usar la instancia de ese personaje al entrar a la escena del gestor de personaje.

El sistema de guardado se basa en un sistema de ranuras, donde se permite tener hasta 4 personajes guardados. Al terminar el proceso de creación, la aplicación genera un directorio que hace referencia a la ranura seleccionada para crear dicho personaje y dentro guarda en forma de un archivo .json todos sus datos. El momento de guardar un

personaje ocurre al cambiar de escena, pausar o cerrar la aplicación. En ese momento, el programa guarda los datos del personaje al que el *Singleton* hace referencia en el archivo .json del directorio correcto. Ese directorio es guardado en la unidad de almacenamiento del dispositivo, lo que permite que en caso de desinstalar la aplicación y volverla a instalar, pueda volver a cargar los archivos de personajes anteriormente creados, ya que siempre se encontrarán en el mismo directorio de la unidad de almacenamiento. Como en la aplicación sólo puede usarse y cambiar los datos de un personaje simultáneamente (al que hace referencia el *Singleton*), al cambiar de escena, pausar o cerrar la aplicación, los datos se guardarán exitosamente al personaje correcto.

El borrado de personajes es más sencillo. Al aceptar borrar un personaje, se comprueba cual es la ranura en la que se ha borrado, entonces el programa elimina el respectivo directorio donde se guardaba el archivo .json del personaje.

6. Implementación y pruebas

En el presente apartado se darán las explicaciones de los procesos técnicos usados para implementar el sistema diseñado en la aplicación móvil mediante el motor Unity.

6.1. Sistema de clases del proyecto

Para poder implementar los sistemas y todos los elementos que los acompañan, se han creado varias clases que representan varias de las mecánicas que conforman el sistema de rol Animaia o mecánicas propias de la aplicación.

En Unity, todo script al crearse genera por defecto una clase para contener métodos en los que se implementan las funcionalidades. Como en este proyecto la mayoría de scripts no guardan datos, en este apartado sólo se mostrarán esas clases que contienen esos datos necesarios para representar las mecánicas del sistema de rol o de la aplicación. Ver **Figura 6.1/1**.

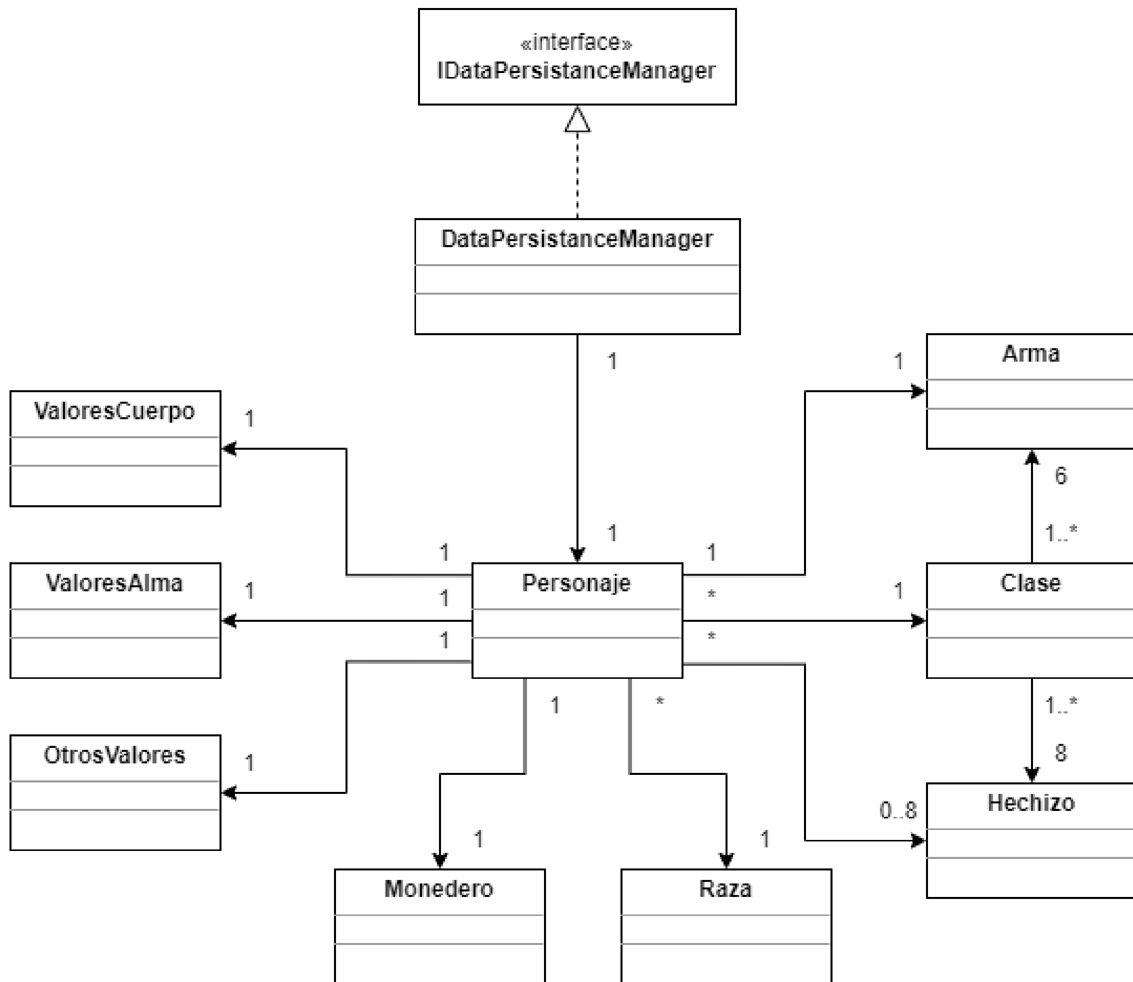

Figura 6.1/1: Diagrama de las clases principales.

6.1.1. Personaje

Es la clase principal, en la que se encapsulan la gran mayoría de clases dentro del proyecto. Sirve para representar, tal como indica el nombre, a un personaje del sistema Animaia. Dentro del proyecto, en ejecución, el Singleton, representado por la clase DataPersistanceManager, hace referencia a esta clase. Ver **Figura 6.1.1/1**.

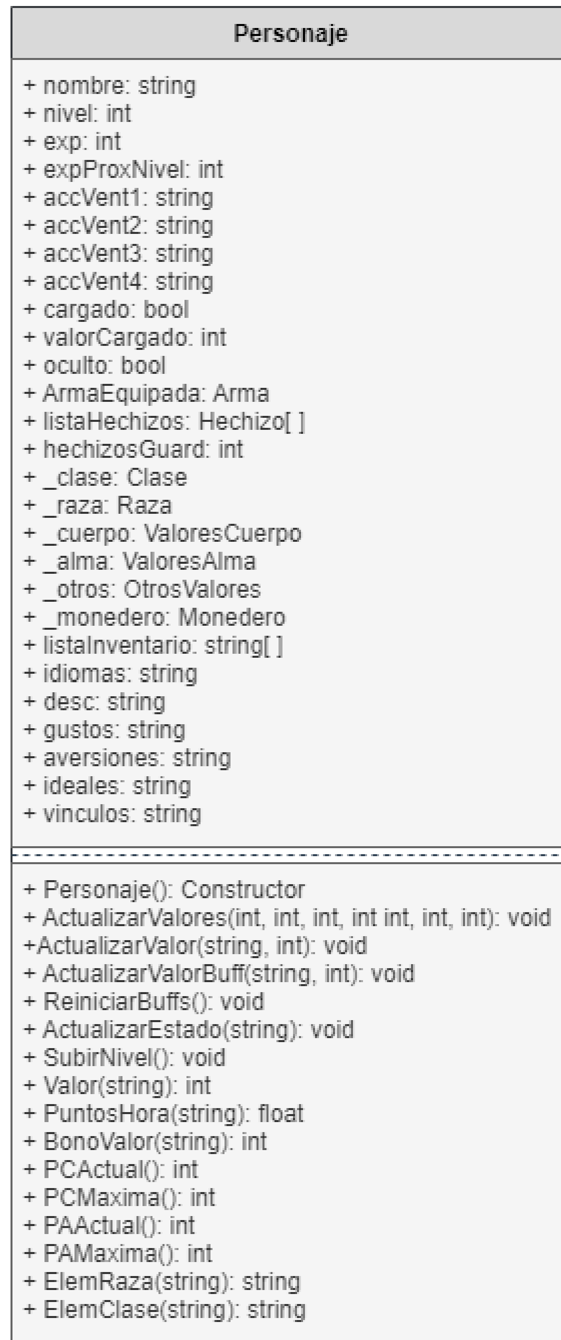

Figura 6.1.1/1: Representación de la clase Personaje.

Esta es la clase que se utiliza en el proyecto para almacenar la gran mayoría de datos necesarios para las funcionalidades de la aplicación. Como el uso de la aplicación se basa en la creación y gestión de personajes, usa esta clase para dichas funcionalidades.

Sus atributos son:

- **nombre:** Nombre del personaje.
- **nivel:** El nivel del personaje.
- **exp:** Puntos de experiencia total acumulados.
- **expProxNivel:** Puntos de experiencia necesarios para subir de nivel.
- **accVent1/accVent2/accVent3/accVent4:** Nombre de las acciones ventaja del personaje.
- **cargado:** Valor booleano para saber si el personaje está bajo los efectos de la habilidad "Carga". Será *true* si lo está. En caso contrario, *false*.
- **valorCargado:** Número del daño adicional que hará el siguiente ataque del personaje.
- **oculto:** Valor booleano para saber si el personaje está bajo los efectos de la habilidad "Mimetizar". Será *true* si lo está. En caso contrario, *false*.
- **ArmaEquipada:** El arma que el personaje tiene equipada, con toda su información.
- **listaHechizos:** Listado de las habilidades que el personaje ha aprendido.
- **hechizosGuard:** Número de las habilidades que el personaje ha aprendido.
- **_clase:** La clase a la que pertenece el personaje, con toda su información.
- **_raza:** La raza a la que pertenece el personaje, con toda su información.
- **_cuerpo:** Los valores corporales del personaje.
- **_alma:** Los valores almáticos del personaje.
- **_otros:** Los otros valores que no son de ninguna de las anteriores categorías; acciones por turno, dados de ataque y estado.
- **_monedero:** Listado ordenado de cada tipo de moneda perteneciente del personaje.
- **listaInventario:** Listado de los objetos que pertenecen al personaje.
- **idiomas:** Texto donde se guardan los idiomas aprendidos por el personaje.
- **desc:** Texto donde se guarda la descripción general del personaje.
- **gustos:** Texto donde se escriben y se guardan las cosas que le gustan al personaje.

- **aversiones:** Texto donde se escriben y se guardan las cosas que no le gustan al personaje.
- **ideales:** Texto donde se escriben y se guardan las ideologías y pensamientos que sigue el personaje.
- **vinculos:** Texto donde se escriben y se guardan las relaciones que tiene el personaje con otros personajes o colectivos.

Sus funciones principales son:

- **ActualizarValores:** Función usada para actualizar todos los valores del personaje a la vez.

```
public void ActualizarValores(int fuerza, int resist, int agilidad, int
                              velocidad, int poder, int sentidos, int
                              memoria, int persona)
{
    _cuerpo.ActValoresCuerpo(fuerza, resist, agilidad, velocidad);
    _alma.ActValoresAlma(poder, sentidos, memoria, persona);
    _otros.apt = 2 + (_cuerpo.bono_velocidad/2);
    _otros.numDados = 1 + (nivel/2);
}
```

Como se puede observar, la función llama a las respectivas funciones de cada tipo de valor para actualizarlos con los parámetros entrados y actualiza los otros 2 tipos de valor existentes, las acciones por turno y el número de dados de ataque.

- **ActualizarValor:** Función que se limita a actualizar el valor especificado en su parámetro de entrada "nombreValor" por el número especificado en el parámetro "numero". Esta función permite la actualización de un valor base en específico, por ejemplo al subir de nivel.

  Lo primero que hace la función es comprobar en una cadena de ifs cuál es el tipo de valor a actualizar. Cuando encuentra el valor, entonces llama a su función específica pasándole el número del valor ya actualizado. En caso de ser el cuerpo o el alma, como no tienen funciones específicas, el cambio se hace accediendo directamente a los valores.

- **ActualizarValorBuff:** Similar al anterior función, pero actualizando los valores con cambios temporales. Eso significa que se usa solo cuando al personaje se le aplican buffs o debuffs que se revertirán después.

  Su funcionamiento es casi idéntico al de ActualizarValor, pero sólo aplica cambios en los valores que pueden ser alterados temporalmente llamando a las funciones de cada valor dedicadas a ese cometido.

- **ReiniciarBuffs():** Función usada para eliminar los buffs y debuffs del personaje.

```
public void ReiniciarBuffs()
{
    _cuerpo.ReiniciarValoresCuerpo();
    _alma.ReiniciarValoresAlma();
}
```

Esta función llama a las funciones de los valores físicos y almáticos para que vuelvan a sus valores numéricos base.

- **SubirNivel():** Función usada cuando el personaje sube de nivel. Actualiza sus valores base, su nivel y su número de dados de ataque disponibles.

```
public void SubirNivel()
{
    _cuerpo.extra_vida = 8 * nivel;
    _alma.extra_mana = 8 * nivel;
    _cuerpo.ActValoresCuerpo(_cuerpo.fuerza, _cuerpo.resist,
                             _cuerpo.agilidad, _cuerpo.velocidad);
    _alma.ActValoresAlma(_alma.poder, _alma.sentidos, _alma.memoria,
                         _alma.persona);
    nivel++;
    _otros.numDados = 1 + (nivel / 2);
}
```

- **Valor():** Función que sirve para conseguir el valor solicitado por su nombre, pasado como parámetro de la función.

La función comprueba cuál es el valor solicitado por el nombre que guarda la variable “nombreValor”. Al encontrarlo, devuelve el valor numérico actual del valor solicitado.

- **PuntosHora():** Función que devuelve los puntos que recupera el personaje por hora del valor con el nombre que haya sido pasado como parámetro de la función. Solo puede comprobar el cuerpo o el alma y es usada cuando el personaje hace un descanso.

```
public float PuntosHora(string nombreValor)
{
    float res = 0.0f;
    if (nombreValor == "CUERPO") { res = _cuerpo.cuerpo_hora; }
    else if (nombreValor == "ALMA") { res = _alma.alma_hora; }
    return res;
}
```

En esta función primero comprueba el nombre del valor guardado en la variable “nombreValor”. Si ese valor es “CUERPO” o “ALMA”, devuelve sus respectivos puntos por hora.

- **BonoValor():** Similar a la función Valor(), esta busca el valor con el nombre entrado en el parámetro “nombreValor”, y devuelve el bono asociado a ese valor.

- **PCActual():** Función usada para obtener los puntos de cuerpo actuales del personaje; su vida actual.

```
public int PCActual()
{
    int res = _cuerpo.cuerpo_act;
    return res;
}
```

- **PCMaxima():** Función usada para obtener los puntos de cuerpo máximos del personaje; su vida total.

```
public int PCMaxima()
{
    int res = _cuerpo.cuerpo_max;
    return res;
}
```

- **PAActual():** Función usada para obtener los puntos de alma actuales del personaje; su maná actual.

```
public int PAActual()
{
    int res = _alma.alma_act;
    return res;
}
```

- **PAMaxima():** Función usada para obtener los puntos de alma máximos del personaje; su maná total.

```
public int PAMaxima()
{
    int res = _alma.alma_max;
    return res;
}
```

- **ElemRaza():** Función usada para obtener el tipo de información de la raza solicitada por el parámetro de la función "elem". Esta función llama a la función GetElemRaza de la clase Raza y le pasa el parámetro "elem". Esta última función devuelve la información solicitada.

```
public string ElemRaza(string elem)
{
    return _raza.GetElemRaza(elem);
}
```

- **ElemClase():** Función usada para obtener el tipo de información de la clase solicitada por el parámetro de la función "elem". Esta función llama a la función GetElemClase de la clase Clase y le pasa el parámetro "elem". Esta última función devuelve la información solicitada.

```
public string ElemClase(string elem)
{
    return _clase.GetElemClase(elem);
}
```

6.1.2. Data Persistance Manager

Clase usada como el Singleton de la aplicación. Permite la manipulación de los personajes durante la ejecución, ya sea cargando los datos de los personajes, guardándolos, creando nuevos personajes o su borrado. Ver **Figura 6.1.2/1**.

Figura 6.1.2/1: Representación de la clase DataPersistanceManager.

Esta clase siempre hace referencia a un personaje en concreto. Cuando el usuario decide crear uno nuevo entrando en el creador de personajes, esta es la clase que se dedica a crearlo, haciendo referencia a un personaje vacío. Cuando termina el proceso del creador de personajes, esta clase guarda los datos del nuevo personaje en un fichero .json, lo que permite su uso después de cerrar la aplicación. Si el usuario cambia de

personaje, esta clase se dedica a consultar el fichero del otro personaje y cargar sus datos en su instancia de personaje, completando así el cambio de personaje. De esta forma, durante la ejecución de la aplicación, siempre se sabe de cuál personaje se debe consultar y editar sus datos, manteniendo su instancia entre escenas.

Sus atributos son:

- **initializeDataIfNull:** Atributo booleano que permite la ejecución de la aplicación sin hacer referencia a un Personaje. Usado para debuggear, en ejecución real no tiene uso.
- **filename:** nombre que reciben los archivos de guardado de los personajes. En ejecución, este atributo guarda el nombre "personaje.pers".
- **instance:** Instancia de sí mismo. Encargada de comprobar que el Singleton funcione correctamente, sirve para revisar si la instancia de la escena es la misma que a la que hace referencia.
- **Personaje:** Representa al personaje actual, al que la aplicación debe consultar y editar sus datos en ese momento.
- **dataPersistancePers:** Listado de las interfaces de esta clase activas en la escena actual.
- **dataHandler:** Referencia al script usado para poder cargar, guardar o eliminar los datos del personaje al que se hace referencia.
- **idPersSeleccionado:** Código de identificación que recibe el Personaje usado durante la ejecución. Este código permite al dataHandler saber en qué fichero manipular los datos.
- **idPersGuardado:** Sirve para mantener el id del personaje actual en caso que la aplicación pase a un segundo plano, ya que al ocurrir esto durante la fase de creación de personajes, el "idPersSeleccionado" pierde el código del personaje actual.
- **isCreating:** Atributo booleano que se utiliza para conocer si la aplicación está en la escena del creador de personajes o no. Si lo está, su valor será *true*, de lo contrario, será *false*.

- **isPaused:** Atributo booleano que se utiliza para conocer si la aplicación está en segundo plano o no. Si lo está, su valor será *true*, de lo contrario, será *false*.

Sus funciones son:

- **Awake():** Función que se ejecuta cuando se inicia una escena y asegura el correcto funcionamiento del Singleton. Comprueba que no haya otra instancia de esta clase en la escena. Si ese es el caso, la destruye, ya que solo puede haber una. Luego vincula al atributo "instance" la instancia actual, se asegura de que no se elimine al cambiar de escenas y crea un DataHandler con el directorio donde manipular los archivos de los personajes y el nombre que deben recibir esos archivos.

```
private void Awake()
{
    if (instance != null)
    {
        Debug.Log("ERROR. Se ha encontrado mas de una instancia en la
                  escena. Destruyendo la nueva...");
        Destroy(this.gameObject);
        return;
    }
    instance = this;
    DontDestroyOnLoad(this.gameObject);

    this.dataHandler = new FileDataHandler(Application.persistentDataPath,
                                           filename);
}
```

- **OnEnable():** Función que se ejecuta cuando el objeto se activa en la escena. Sirve para delegar la función "OnSceneLoaded" a la función de cargar escenas propia del motor Unity, llamada "sceneLoaded". De esta forma, al cargar una escena mediante el uso de las funciones de Unity, "OnSceneLoaded" también se ejecutará.

```
private void OnEnable()
{
    SceneManager.sceneLoaded += OnSceneLoaded;
}
```

- **OnDisable():** Función que se ejecuta cuando el objeto se desactiva en la escena. Tiene el uso contrario a la función anterior, desvincular la función "OnSceneLoaded" a la función de Unity "sceneLoaded".

```
private void OnDisable()
{
    SceneManager.sceneLoaded -= OnSceneLoaded;
}
```

- **OnSceneLoaded():** Función usada para inicializar el atributo "dataPersistancePers", llamando la función "FindAllDataPersistancePers". También carga el Personaje asignado a la clase llamando la función "CargarPersonaje".

```
public void OnSceneLoaded(Scene scene, LoadSceneMode mode)
{
    Debug.Log("Usando OnSceneLoaded...");
    this.dataPersistancePers = FindAllDataPersistancePers();
    CargarPersonaje();
}
```

- **CrearPersonaje():** Función que genera una instancia nueva de la clase Personaje y la asigna al atributo "Personaje".

```
public void CrearPersonaje()
{
    this.Personaje = new Personaje();
}
```

- **GuardarPersonaje():** Función usada para guardar los datos del Personaje asociado a la variable "Personaje". En caso de que no haya un Personaje asociado, lanza un mensaje de aviso y termina su ejecución. Para cada instancia guardada en el listado de la interfaces de esta clase que haya en la escena, el atributo "dataPersistancePers", llama a la función "GuardarData", y luego llama a la función del atributo "dataHandler" llamada "Guardar", pasándole como parámetros el

Personaje activo y su id guardado en el atributo "idPersSeleccionado".

```
    public void GuardarPersonaje()
    {
        if(this.Personaje == null)
        {
            Debug.LogWarning("No se han encontrado personajes. Se necesita crear
                             uno antes");
            return;
        }
        foreach (IDataPersistance dataPersistanceObj in dataPersistancePers)
        {
            dataPersistanceObj.GuardarData(Personaje);
        }
        dataHandler.Guardar(Personaje, idPersSeleccionado);
    }
```

- **CargarPersonaje():** Función usada para cargar los datos del Personaje activo. Esta función llama la función del "dataHandler" llamada "Cargar", pasándole como parámetro el atributo "idPersSeleccionado" para asociar los datos cargados al Personaje del atributo "Personaje". Luego comprueba si ese Personaje existe o no. Si no existe, lo crea ejecutando la función "CrearPersonaje". Luego, por cada instancia de interfaz guardada en "dataPersistanceObj", llama su función "CargarData" para que todos los scripts con la interfaz de esta clase puedan manipular los datos del personaje actual.

```
public void CargarPersonaje()
    {
        this.Personaje = dataHandler.Cargar(idPersSeleccionado);


        if(this.Personaje == null)
        {
            CrearPersonaje();
            Debug.Log("No se ha encontrado el personaje. Creando uno...");
            return;
        }
        foreach (IDataPersistance dataPersistanceObj in dataPersistancePers)
        {
            dataPersistanceObj.CargarData(Personaje);
        }
    }
```

- **ActualizarPersonaje():** Función usada para actualizar los datos del Personaje actual. Lo que hace esta función es asignar el Personaje pasado por parámetro en esta función a la variable de la clase “Personaje”. Entonces es cuando llama a las funciones “GuardarPersonaje” y “CargarPersonaje”, para que esos datos del Personaje se guarden en su respectivo fichero, y para evitar errores, se cargan de nuevo esos datos desde el fichero.

```
public void ActualizarPersonaje(Personaje pers)
{
    this.Personaje = pers;
    GuardarPersonaje();
    CargarPersonaje();
}
```

- **EliminarPersonaje():** Para borrar personajes, se usa esta función. Llama a la función “Eliminar” del “dataHandler”, pasándole por referencia el id del Personaje a eliminar para que elimine el fichero correcto. Después, como ese personaje ya no existe, la clase deja de tener asignado uno, así que cambia el valor de “idPersSeleccionado” a “test”.

```
public void EliminarPersonaje(string persId)
{
    dataHandler.Eliminar(persId);
    this.idPersSeleccionado = "test";
}
```

- **OnApplicationQuit():** Función usada al cerrar la aplicación. Esta función se dedica a guardar el personaje actual llamando la función “GuardarPersonaje”. Sólo lo guarda si el Personaje actual tiene un id asignado. En caso que el Personaje de la variable “Personaje” tenga su nombre vacío, significa que el usuario ha cerrado la aplicación en mitad del proceso de creación, por lo tanto no es necesario guardarlo y lo elimina usando la función “EliminarPersonaje”. Si “idPersSeleccionado” guarda el nombre “test”, también elimina los datos guardados en el fichero del personaje con id “test”.

```
private void OnApplicationQuit()
{
    if(idPersSeleccionado != "test")
        GuardarPersonaje();
    if(this.Personaje.nombre == " ")
    {
        EliminarPersonaje(idPersSeleccionado);
    }
    EliminarPersonaje("test");
}
```

- **OnApplicationPause():** Función usada al dejar la aplicación en segundo plano y al volverla a tener en primer plano. Al ejecutarse, cambia el valor del atributo “isPaused” a *true* si la aplicación se pausa, o *false* si la aplicación se reanuda. Luego comprueba el estado de ese atributo. Si la aplicación se pausa, guarda los datos del Personaje actual con la función “GuardarPersonaje”. Si esa pausa ocurre durante el proceso de creación, comprobado por el estado de la variable “isCreating”, guarda el Personaje actual asignando como id el código “test” usando la función “Guardar” del “dataHandler” y usa la función “EliminarPersonaje”. Esto se hace porque en Android, al pausar una aplicación, el usuario puede cerrarla, pero al estar pausada, no se ejecutaría la función “OnApplicationQuit”, así que, por si acaso, se elimina.

```
 private void OnApplicationPause(bool pauseStatus)
{
    isPaused = pauseStatus;
    Debug.Log("Pausa cambiado a " + isPaused);
    if (isPaused)
    {
        Debug.Log("Se supone que salgo");
        if (isCreating)
        {
            dataHandler.Guardar(Personaje, "test");
            EliminarPersonaje(idPersSeleccionado);
        }
        else
        {
            GuardarPersonaje();
        }
    }
```

Si la aplicación vuelve al primer plano, carga de nuevo el Personaje activo antes de la pausa llamando la función "CargarPersonaje". Si antes se estaba en el proceso de creación de personaje, como se habrá eliminado el personaje, el atributo "idPersSeleccionado" habrá pasado a ser "test", pero como antes se ha guardado el Personaje con ese mismo id, lo cargará de nuevo; es entonces cuando "idPersSeleccionado" pasa a ser el código que tenia antes, ya que se le asigna el que hay guardado en "idPersGuardado" y en caso que antes estuviera creando un personaje, lo guarda de nuevo, pero con el código de identificación correcto.

```
    else
    {
        Debug.Log("He vuelto");
        CargarPersonaje();
        idPersSeleccionado = idPersGuardado;
        if (isCreating)
        {
            GuardarPersonaje();
        }
    }
}
```

- **FindAllDataPersistancePers():** Función usada para inicializar el atributo de la clase "dataPersistancePers". Lo

que hace esta función es buscar todas las interfaces de esta clase activas en la escena actual y las guarda en una lista.

```
private List<IDataPersistance> FindAllDataPersistancePers()
{
    IEnumerable<IDataPersistance> dataPersistancePers =
    FindObjectsOfType<MonoBehaviour>().OfType<IDataPersistance>();

    return new List<IDataPersistance>(dataPersistancePers);
}
```

- **PersonajeExiste():** Función que comprueba si el atributo de la clase "Personaje" está vacío o no. Devuelve *true* si no lo está, pero del caso contrario, devuelve *false*.

```
public bool PersonajeExiste()
{
    return Personaje != null;
}
```

- **CargarPerfilesPersonaje():** Función usada para cargar los Personajes guardados anteriormente en ficheros. Para hacerlo, llama a la función "CargarPerfiles" del atributo "dataHandler". Devuelve un diccionario de cada Personaje guardado con su respectivo código de identificación como clave.

```
public Dictionary<string, Personaje> CargarPerfilesPersonaje()
{
    return dataHandler.CargarPerfiles();
}
```

- **CambiarIdPersSelecc():** Función que asocia el parámetro pasado "IdNuevoPers" a los atributos "idPersSeleccionado" y "idPersGuardado" al momento de seleccionar una de las ranuras de personaje en el menú inicial. También carga sus datos usando la función "CargarPersonaje".
-

```
public void CambiarIdPersSelecc(string IdNuevoPers)
{
    this.idPersSeleccionado = IdNuevoPers;
    this.idPersGuardado = IdNuevoPers;
    CargarPersonaje();
}
```

- **IdPersAct():** Función usada para obtener el código de identificación del Personaje actual. Devuelve el valor del atributo "idPersSeleccionado".

```
public string IdPersAct()
{
    return this.idPersSeleccionado;
}
```

- **ChangeIsCreating():** Función usada para cambiar el estado del atributo "isCreating" por el valor de la variable pasada por parámetro "change".

```
public void ChangeIsCreating(bool change)
{
    isCreating = change;
}
```

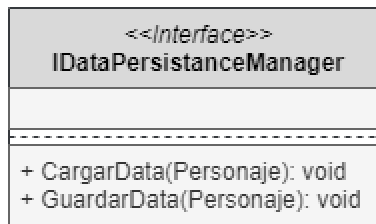

Figura 6.1.2/2: Repesentación de la interfaz de la clase DataPersistanceManager.

Como se ha comentado, esta clase tiene una interfaz que usan el resto de scripts para poder acceder a los datos del Personaje activo, al que hace referencia el Singleton. Esta se llama IDataPersistance. Ver **Figura 6.1.1/2**. Sus funciones son:

- **CargarData():** Asocia el Personaje del Singleton, pasado por referencia, como el Personaje del script.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}
```

- **GuardarData():** Asocia el Personaje del script como el Personaje del Singleton, pasado por referencia.

```
public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

6.1.3. ValoresCuerpo

Clase que sirve para representar los valores corporales de un personaje y sus bonos. Ver **Figura 6.1.3/1**.

Figura 6.1.3/1: Repesentación de la clase ValoresCuerpo.

El uso de esta clase se limita a almacenar los datos del personaje vinculados a los valores corporales y proporcionar métodos de consulta y modificación de esos datos.

Sus atributos son:

- **fuerza:** Valor numérico que representa la fuerza base del personaje.
- **resist:** Valor numérico que representa la resistencia física base del personaje.
- **agilidad:** Valor numérico que representa la agilidad base del personaje.
- **velocidad:** Valor numérico que representa la velocidad base del personaje.
- **extra_vida:** Valor numérico que representan los puntos de vida que se agregan a los puntos de vida máximos al subir de nivel.
- **bono_fuerza:** Valor numérico que representa el bono de fuerza del personaje.
- **bono_resist:** Valor numérico que representa el bono de resistencia del personaje.
- **bono_agilidad:** Valor numérico que representa el bono de agilidad del personaje.
- **bono_velocidad:** Valor numérico que representa el bono de velocidad del personaje.
- **fuerzaAct:** Valor numérico que representa el valor actual de fuerza del personaje.
- **resistAct:** Valor numérico que representa el valor actual de resistencia del personaje.
- **agilidadAct:** Valor numérico que representa el valor actual de agilidad del personaje.
- **velocidadAct:** Valor numérico que representa el valor actual de velocidad del personaje.
- **cuerpo_act:** Valor numérico que representa la vida actual del personaje. Los puntos de cuerpo actuales.
- **cuerpo_max:** Valor numérico que representa la vida máxima del personaje, los puntos de cuerpo totales.
- **cuerpo_hora:** Valor numérico que representan los puntos de vida que el personaje recupera por hora descansada.

Sus funciones son:

- **ActValoresCuerpo():** Función para actualizar todos los valores físicos a la vez. Sustituye cada valor por su equivalente entrado como parámetro de la función. Luego de sustituir cada valor, tanto base como actual, actualiza los respectivos bonos, calculándolos de nuevo. Finalmente recalcula los puntos de vida máximos del personaje, los puntos de vida actuales pasan a ser los mismos que los puntos máximos y finalmente se recalcula el atributo "cuerpo_hora".

```
 public void ActValoresCuerpo(int fuerza, int resist, int agilidad, int
                              velocidad)
{
    this.fuerza = fuerza;
    this.resist = resist;
    this.agilidad = agilidad;
    this.velocidad = velocidad;
    this.fuerzaAct = fuerza;
    this.resistAct = resist;
    this.agilidadAct = agilidad;
    this.velocidadAct = velocidad;

    if (fuerzaAct >= 8) { this.bono_fuerza = (fuerzaAct - 8) / 2; }
    else { this.bono_fuerza = (fuerzaAct - 9) / 2; }
    if (resistAct >= 8) { this.bono_resist = (resistAct - 8) / 2; }
    else { this.bono_resist = (resistAct - 9) / 2; }
    if (agilidadAct >= 8) { this.bono_agilidad = (agilidadAct - 8) / 2; }
    else { this.bono_agilidad = (agilidadAct - 9) / 2; }
    if (velocidadAct >= 8) { this.bono_velocidad = (velocidadAct - 8) / 2; }
    else { this.bono_velocidad = (velocidadAct - 9) / 2; }

    this.cuerpo_max = 20 + (bono_resist * 2) + extra_vida;
    this.cuerpo_act = cuerpo_max;
    this.cuerpo_hora = (cuerpo_max * 10.0f) / 100.0f;
}
```

- **ActFuerza():** Función usada para actualizar únicamente el valor base de la fuerza. Esta función primero suma el valor del atributo "fuerza" y "fuerzaAct" con el número pasado por referencia llamado "numero". Hecho eso, actualiza el atributo "bono_fuerza" recalculando el bono de fuerza.

```
public void ActFuerza(int numero)
{
    fuerza = fuerza + numero;
    fuerzaAct = fuerzaAct + numero;
    if (fuerzaAct >= 8) { bono_fuerza = (fuerzaAct - 8) / 2; }
    else { bono_fuerza = (fuerzaAct - 9) / 2; }
}
```

- **ActFuerzaBuff():** Similar a la función anterior, pero sólo actualizando el valor actual del valor fuerza, ya que se suele usar sólo cuando se aplican cambios de valores temporales. Esta función primero suma el valor del atributo "fuerzaAct" con el número pasado por referencia llamado "numero". Hecho eso, actualiza el atributo "bono_fuerza" recalculando el bono de fuerza.

```
public void ActFuerzaBuff(int numero)
{
    fuerzaAct = fuerzaAct + numero;
    if (fuerzaAct >= 8) { bono_fuerza = (fuerzaAct - 8) / 2; }
    else { bono_fuerza = (fuerzaAct - 9) / 2; }
}
```

- **ActResis():** Función usada para actualizar únicamente el valor base de la resistencia. Esta función primero suma el valor del atributo "resistencia" y "resistAct" con el número pasado por referencia, llamado "numero". Hecho eso, actualiza el atributo "bono_resist" recalculando el bono de resistencia. Finalmente, actualiza el atributo "cuerpo_max" con el cálculo de puntos de vida del personaje, el atributo "cuerpo_act" pasa a tener el mismo valor que "cuerpo_max" y por último actualiza el atributo "cuerpo_hora" con su cálculo respectivo.

```
public void ActResis(int numero)
{
    resist = resist + numero;
    resistAct = resistAct + numero;
    if (resistAct >= 8) { bono_resist = (resistAct - 8) / 2; }
    else { bono_resist = (resistAct - 9) / 2; }
    cuerpo_max = 20 + (bono_resist * 2) + extra_vida;
    cuerpo_act = cuerpo_max;
    cuerpo_hora = (cuerpo_max * 10.0f) / 100.0f;
}
```

- **ActResisBuff():** Similar a la función anterior, pero sólo actualizando el valor actual del valor resistencia, ya que se suele usar sólo cuando se aplican cambios de valores temporales. Esta función primero suma el valor del atributo "resistAct" con el número pasado por referencia llamado "numero". Hecho eso, actualiza el atributo "bono_resist" recalculando el bono de resistencia. No cambia el atributo "cuerpo_max" porque este está vinculado al valor base de la resistencia, y en esta función se cambia el valor variable.

```
public void ActResisBuff(int numero)
{
    resistAct = resistAct + numero;
    if (resistAct >= 8) { bono_resist = (resistAct - 8) / 2; }
    else { bono_resist = (resistAct - 9) / 2; }
}
```

- **ActAgil():** Función usada para actualizar únicamente el valor base de la agilidad. Esta función primero suma el valor del atributo "agilidad" y "agilidadAct" con el número pasado por referencia llamado "numero". Hecho eso, actualiza el atributo "bono_agilidad" recalculando el bono de agilidad.

```
public void ActAgil(int numero)
{
    agilidad = agilidad + numero;
    agilidadAct = agilidadAct + numero;
    if (agilidadAct >= 8) { bono_agilidad = (agilidadAct - 8) / 2; }
    else { bono_agilidad = (agilidadAct - 9) / 2; }
}
```

- **ActAgilBuff():** Similar a la función anterior, pero sólo actualizando el valor actual del valor agilidad, ya que se suele usar sólo cuando se aplican cambios de valores temporales. Esta función primero suma el valor del atributo "agilidadAct" con el número pasado por referencia llamado "numero". Hecho eso, actualiza el atributo "bono_agilidad" recalculando el bono de agilidad.

```
public void ActAgilBuff(int numero)
{
    agilidadAct = agilidadAct + numero;
    if (agilidadAct >= 8) { bono_agilidad = (agilidadAct - 8) / 2; }
    else { bono_agilidad = (agilidadAct - 9) / 2; }
}
```

- **ActVel():** Función usada para actualizar únicamente el valor base de la velocidad. Esta función primero suma el valor del atributo "velocidad" y "velocidadAct" con el número pasado por referencia llamado "numero". Hecho eso, actualiza el atributo "bono_velocidad" recalculando el bono de velocidad.

```
public void ActVel(int numero)
{
    velocidad = velocidad + numero;
    velocidadAct = velocidadAct + numero;
    if (velocidadAct >= 8) { bono_velocidad = (velocidadAct - 8) / 2; }
    else { bono_velocidad = (velocidadAct - 9) / 2; }
}
```

- **ActVelBuff():** Similar a la función anterior, pero sólo actualizando el valor actual del valor velocidad, ya que se suele usar sólo cuando se aplican cambios de valores temporales. Esta función primero suma el valor del atributo "velocidadAct" con el número pasado por referencia llamado "numero". Hecho eso, actualiza el atributo "bono_velocidad" recalculando el bono de velocidad.

```
public void ActVelBuff(int numero)
{
    velocidadAct = velocidadAct + numero;
    if (velocidadAct >= 8) { bono_velocidad = (velocidadAct - 8) / 2; }
    else { bono_velocidad = (velocidadAct - 9) / 2; }
}
```

- **ReiniciarValoresCuerpo():** Función usada para restaurar los valores corporales actuales y devolverlos a los valores base del personaje. Simplemente sustituye los valores actuales y les vincula el valor numérico de los respectivos valores base. Luego actualiza todos los bonos de valores físicos.

```
public void ReiniciarValoresCuerpo()
{
    fuerzaAct = fuerza;
    resistAct = resist;
    agilidadAct = agilidad;
    velocidadAct = velocidad;

    if (fuerzaAct >= 8) { this.bono_fuerza = (fuerzaAct - 8) / 2; }
    else { this.bono_fuerza = (fuerzaAct - 9) / 2; }
    if (resistAct >= 8) { this.bono_resist = (resistAct - 8) / 2; }
    else { this.bono_resist = (resistAct - 9) / 2; }
    if (agilidadAct >= 8) { this.bono_agilidad = (agilidadAct - 8) / 2; }
    else { this.bono_agilidad = (agilidadAct - 9) / 2; }
    if (velocidadAct >= 8) { this.bono_velocidad = (velocidadAct - 8) / 2; }
    else { this.bono_velocidad = (velocidadAct - 9) / 2; }
}
```

6.1.4. ValoresAlma

Esta clase sirve para representar los valores almáticos de un personaje y los bonos de estos. Ver **Figura 6.1.4/1**.

Figura 6.1.4/1: Repesentación de la clase ValoresAlma.

El uso de esta clase se limita a almacenar los datos del personaje vinculados a los valores almáticos y proporcionar métodos de consulta y modificación de esos datos.

Sus atributos son:

- **poder:** Valor numérico que representa el poder base del personaje.
- **sentidos:** Valor numérico que representa los sentidos base del personaje.
- **memoria:** Valor numérico que representa la memoria base del personaje.
- **persona:** Valor numérico que representa la personalidad base del personaje.
- **extra_mana:** Valor numérico que representan los puntos de alma que se agregan a los puntos de alma máximos al subir de nivel.
- **bono_poder:** Valor numérico que representa el bono de poder del personaje.

- **bono_sentidos:** Valor numérico que representa el bono de sentidos del personaje.
- **bono_memoria:** Valor numérico que representa el bono de memoria del personaje.
- **bono_persona:** Valor numérico que representa el bono de personalidad del personaje.
- **poderAct:** Valor numérico que representa el valor actual de poder del personaje.
- **sentidosAct:** Valor numérico que representa el valor actual de los sentidos del personaje.
- **memoriaAct:** Valor numérico que representa el valor actual de memoria del personaje.
- **personaAct:** Valor numérico que representa el valor actual de personalidad del personaje.
- **alma_act:** Valor numérico que representa los puntos de alma actuales del personaje..
- **alma_max:** Valor numérico que representa los puntos de alma máximos del personaje. Los puntos de alma totales.
- **alma_hora:** Valor numérico que representan los puntos de alma que el personaje recupera por hora descansada.

Sus funciones son:

- **ActValoresAlma():** Función para actualizar todos los valores almáticos a la vez. Sustituye cada valor por su equivalente entrado como parámetro de la función. Luego de sustituir cada valor, tanto base como actual, actualiza los respectivos bonos, calculándolos de nuevo. Finalmente recalcula los puntos de alma máximos del personaje, los puntos de alma actuales pasan a ser los mismos que los puntos máximos y finalmente se recalcula el atributo “alma_hora”.

```
public void ActValoresAlma(int poder, int sentidos, int memoria, int
                           persona)
{
    this.poder = poder;
    this.sentidos = sentidos;
    this.memoria = memoria;
    this.persona = persona;
    this.poderAct = poder;
    this.sentidosAct = sentidos;
```

```
        this.memoriaAct = memoria;
        this.personaAct = persona;

        if (poderAct >= 8) { this.bono_poder = (poderAct - 8) / 2; }
        else { this.bono_poder = (poderAct - 9) / 2; }
        if (sentidosAct >= 8) { this.bono_sentidos = (sentidosAct - 8) / 2; }
        else { this.bono_sentidos = (sentidosAct - 9) / 2; }
        if (memoriaAct >= 8) { this.bono_memoria = (memoriaAct - 8) / 2; }
        else { this.bono_memoria = (memoriaAct - 9) / 2; }
        if (personaAct >= 8) { this.bono_persona = (personaAct - 8) / 2; }
        else { this.bono_persona = (personaAct - 9) / 2; }

        this.alma_max = 30 + (bono_poder * 2) + extra_mana;
        this.alma_act = alma_max;
        this.alma_hora = (alma_max * 10.0f) / 100.0f;
    }
```

- **ActPoder():** Función usada para actualizar únicamente el valor base de poder. Esta función primero suma el valor del atributo "poder" y "poderAct" con el número pasado por referencia llamado "numero". Hecho eso, actualiza el atributo "bono_poder" recalculando el bono de poder. Finalmente, actualiza el atributo "alma_max" con el cálculo de puntos de alma del personaje, el atributo "alma_act" pasa a tener el mismo valor que "alma_max" y por último actualiza el atributo "alma_hora" con su cálculo respectivo.

```
    public void ActPoder(int numero)
    {
        poder = poder + numero;
        poderAct = poderAct + numero;
        if (poderAct >= 8) { bono_poder = (poderAct - 8) / 2; }
        else { bono_poder = (poderAct - 9) / 2; }
        alma_max = 30 + (bono_poder * 2) + extra_mana;
        alma_act = alma_max;
        alma_hora = (alma_max * 10.0f) / 100.0f;
    }
```

- **ActPoderBuff():** Similar a la función anterior, pero sólo actualizando el valor actual del valor poder, ya que se suele usar sólo cuando se aplican cambios de valores temporales. Esta función primero suma el valor del atributo "poderAct" con el número pasado por referencia llamado "numero". Hecho eso, actualiza el atributo

“bono_poder” recalculando el bono de poder. No cambia el atributo “poder_max” porque este está vinculado al valor base del poder, y en esta función solo se cambia el valor variable.

```
public void ActPoderBuff(int numero)
{
    poderAct = poderAct + numero;
    if (poderAct >= 8) { bono_poder = (poderAct - 8) / 2; }
    else { bono_poder = (poderAct - 9) / 2; }
}
```

- **ActSent():** Función usada para actualizar únicamente el valor base de los sentidos. Esta función primero suma el valor del atributo “sentidos” y “sentidosAct” con el número pasado por referencia llamado “numero”. Hecho eso, actualiza el atributo “bono_sentidos” recalculando el bono de sentidos.

```
public void ActSent(int numero)
{
    sentidos = sentidos + numero;
    sentidosAct = sentidosAct + numero;
    if (sentidosAct >= 8) { bono_sentidos = (sentidosAct - 8) / 2; }
    else { bono_sentidos = (sentidosAct - 9) / 2; }
}
```

- **ActSentBuff():** Similar a la función anterior, pero sólo actualizando el valor actual del valor sentidos, ya que se suele usar sólo cuando se aplican cambios de valores temporales. Esta función primero suma el valor del atributo “sentidosAct” con el número pasado por referencia llamado “numero”. Hecho eso, actualiza el atributo “bono_sentidos” recalculando el bono de sentidos.

```
public void ActSentBuff(int numero)
{
    sentidosAct = sentidosAct + numero;
    if (sentidosAct >= 8) { bono_sentidos = (sentidosAct - 8) / 2; }
    else { bono_sentidos = (sentidosAct - 9) / 2; }
}
```

- **ActMemo():** Función usada para actualizar únicamente el valor base de la memoria. Esta función primero suma el valor del atributo “memoria” y “memoriaAct” con el número pasado por referencia llamado “numero”. Hecho eso, actualiza el atributo “bono_memoria” recalculando el bono de memoria.

```
public void ActMemo(int numero)
{
    memoria = memoria + numero;
    memoriaAct = memoriaAct + numero;
    if (memoriaAct >= 8) { bono_memoria = (memoriaAct - 8) / 2; }
    else { bono_memoria = (memoriaAct - 9) / 2; }
}
```

- **ActMemoBuff():** Similar a la función anterior, pero sólo actualizando el valor actual del valor memoria, ya que se suele usar sólo cuando se aplican cambios de valores temporales. Esta función primero suma el valor del atributo “memoriaAct” con el número pasado por referencia llamado “numero”. Hecho eso, actualiza el atributo “bono_memoria” recalculando el bono de memoria.

```
public void ActMemoBuff(int numero)
{
    memoriaAct = memoriaAct + numero;
    if (memoriaAct >= 8) { bono_memoria = (memoriaAct - 8) / 2; }
    else { bono_memoria = (memoriaAct - 9) / 2; }
}
```

- **ActPers():** Función usada para actualizar únicamente el valor base de la personalidad. Esta función primero suma el valor del atributo “persona” y “personaAct” con el número pasado por referencia llamado “numero”. Hecho eso, actualiza el atributo “bono_persona” recalculando el bono de personalidad.

```
public void ActPers(int numero)
{
    persona = persona + numero;
    personaAct = personaAct + numero;
    if (personaAct >= 8) { bono_persona = (personaAct - 8) / 2; }
    else { bono_persona = (personaAct - 9) / 2; }
}
```

- **ActPersBuff():** Similar a la función anterior, pero sólo actualizando el valor actual del valor personalidad, ya que se suele usar sólo cuando se aplican cambios de valores temporales. Esta función primero suma el valor del atributo "personaAct" con el número pasado por referencia llamado "numero". Hecho eso, actualiza el atributo "bono_persona" recalculando el bono de personalidad.

```
public void ActPersBuff(int numero)
{
    personaAct = personaAct + numero;
    if (personaAct >= 8) { bono_persona = (personaAct - 8) / 2; }
    else { bono_persona = (personaAct - 9) / 2; }
}
```

- **ReiniciarValoresAlma():** Función usada para restaurar los valores almáticos actuales y devolverlos a los valores base del personaje. Simplemente sustituye los valores actuales y les vincula el valor numérico de los respectivos valores base. Luego actualiza todos los bonos de valores almáticos.

```
public void ReiniciarValoresAlma()
{
    this.poderAct = poder;
    this.sentidosAct = sentidos;
    this.memoriaAct = memoria;
    this.personaAct = persona;

    if (poderAct >= 8) { this.bono_poder = (poderAct - 8) / 2; }
    else { this.bono_poder = (poderAct - 9) / 2; }
    if (sentidosAct >= 8) { this.bono_sentidos = (sentidosAct - 8) / 2; }
    else { this.bono_sentidos = (sentidosAct - 9) / 2; }
    if (memoriaAct >= 8) { this.bono_memoria = (memoriaAct - 8) / 2; }
    else { this.bono_memoria = (memoriaAct - 9) / 2; }
    if (personaAct >= 8) { this.bono_persona = (personaAct - 8) / 2; }
    else { this.bono_persona = (personaAct - 9) / 2; }
}
```

6.1.5. OtrosValores

Esta clase sirve para representar el resto de valores de personaje que no entran en las categorías anteriores. Ver **Figura 6.1.5/1**.

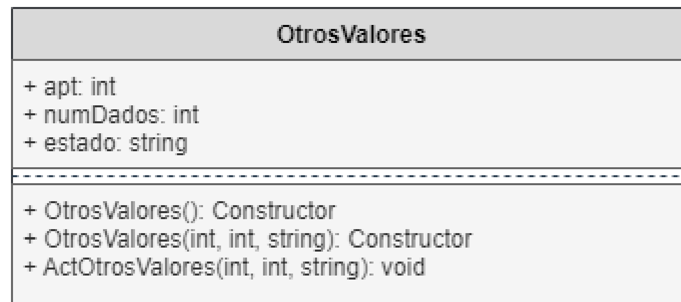

Figura 6.1.5/1: Repesentación de la clase OtrosValores.

El uso de esta clase se limita a almacenar los datos del personaje vinculados a los valores que no están especificados en las 2 clases anteriores.

Sus atributos son:

- **apt:** Representa el valor de Acciones Por Turno.
- **numDados:** Representa el número de dados de ataque que dispone el personaje.

Su única función es ActOtrosValores(). Esta simplemente permite actualizar todos los atributos de la clase sustituyéndolos por los respectivos parámetros de la función equivalentes.

```
public void ActOtrosValores(int apt, int numDados, string
                            estado)
{
    this.apt = apt;
    this.numDados = numDados;
    this.estado = estado;
}
```

6.1.6. Raza

Esta clase representa la raza a la que pertenece el personaje y almacena toda su información. Ver **Figura 6.1.6/1**.

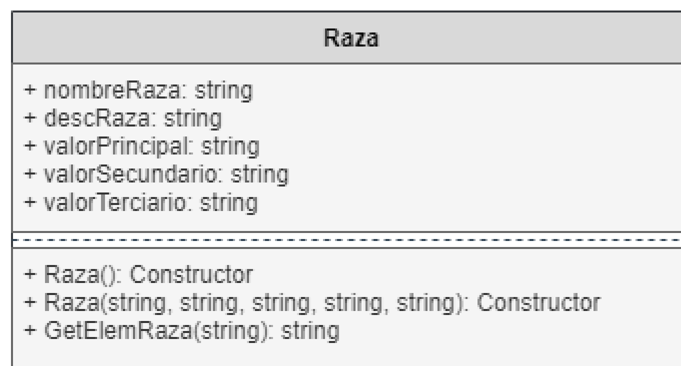

Figura 6.1.6/1: Repesentación de la clase Raza.

El uso de esta clase se limita a guardar los datos de la raza del personaje y proporcionar métodos de consulta de esos datos.

Sus atributos son:

- **nombreRaza:** El nombre de la raza.
- **descRaza:** Una descripción detallada de la raza del personaje.
- **valorPrincipal:** El valor más importante vinculado a la raza, el que ha aumentado en 2 puntos al ser escogida.
- **valorSecundario:** El otro valor fijo asociado de la raza.
- **valorTerciario:** El tercer valor de la raza, que el jugador ha tenido que elegir.

Su única función aparte de los constructores de la clase se llama GetElemRaza. Este método funciona como un getter, en el que se le pasa el nombre del atributo que se quiere consultar como parámetro de la función, llamado "elem". Con ese parámetro y una cadena de ifs, se busca el atributo de la clase que coincida con el valor de "elem" y se devuelve ese atributo en forma de string.

6.1.7. Clase

Esta clase representa la clase a la que pertenece el personaje y almacena toda su información. Ver **Figura 6.1.7/1**.

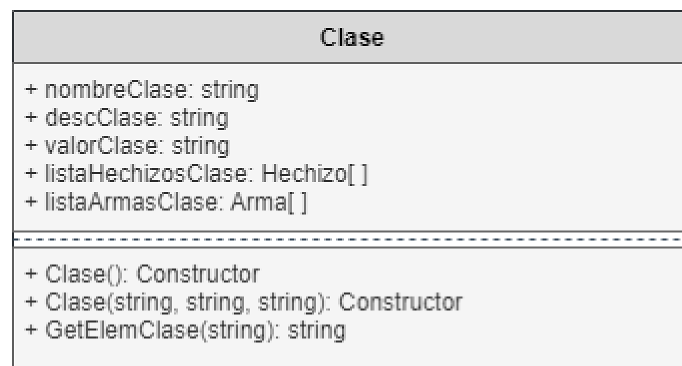

Figura 6.1.7/1: Repesentación de la clase Clase.

El uso de esta clase se limita a guardar los datos de la clase del personaje y proporcionar métodos de consulta de esos datos. Estos datos son los que definen el estilo de juego de ese personaje.

Sus atributos son:

- **nombreClase:** El nombre de la clase del personaje.
- **descClase:** Descripción de la clase del personaje.
- **valorClase:** Valor al que está vinculada la clase y que aumenta en 2 puntos cada vez que el personaje sube de nivel.
- **listaHechizosClase:** Lista de todos las habilidades a los que tiene acceso el personaje por pertenecer a esa clase.
- **listaArmasClase:** Lista de todas las armas que puede escoger el personaje por pertenecer a esa clase.

Sin tener en cuenta los constructores de la clase, esta solo tiene la función GetElemClase. Esta función es un getter en el que se le pasa el nombre del atributo que se quiere consultar como

parámetro de la función, llamado “elem”. Con ese parámetro y una cadena de ifs, se busca el atributo de la clase que coincida con el valor de “elem” y se devuelve ese atributo en forma de string.

```
public string GetElemClase(string elem)
{
    string res = " ";
    if (elem == "nombre") { res = nombreClase; }
    else if (elem == "valor") { res = valorClase; }
    else if (elem == "descripcion") { res = descClase; }
    return res;
}
```

6.1.8. Hechizo

Esta clase representa a una habilidad, nombrada internamente por el nombre original de la mecánica en etapas anteriores de desarrollo. Guarda toda la información necesaria para el correcto funcionamiento de cada habilidad. Ver **Figura 6.1.8/1**.

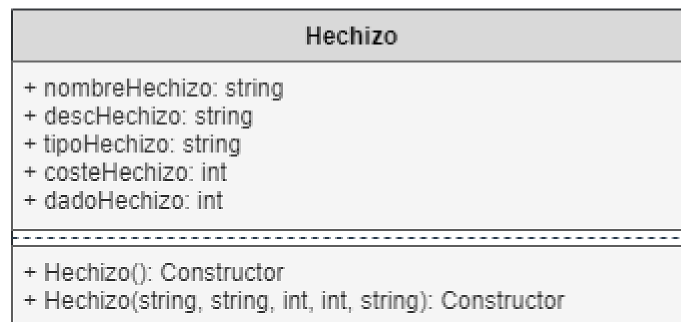

Figura 6.1.8/1: Repesentación de la clase Hechizo.

El uso de esta clase se limita en definir la información que conforma una habilidad dentro de la aplicación.

Sus atributos son:

- **nombreHechizo:** Nombre de la habilidad.
- **descHechizo:** Descripción de lo que hace la habilidad.

- **tipoHechizo:** Nombre del tipo de habilidad que permite identificar su mecánica dentro de la aplicación y la función que se debe ejecutar en caso de usarse.
- **costeHechizo:** Número que simbolizan los puntos de alma que cuestan usar la habilidad.
- **dadoHechizo:** Simboliza el tipo de dado que se usa al ejecutar la habilidad. Si su valor es 0, significa que no se usa una tirada de dados para ejecutar la acción.

Esta clase solo tiene constructores como funciones.

6.1.9. Monedero

Representa el monedero del personaje donde guarda el dinero, separando los diferentes tipos de moneda. Ver **Figura 6.1.9/1**.

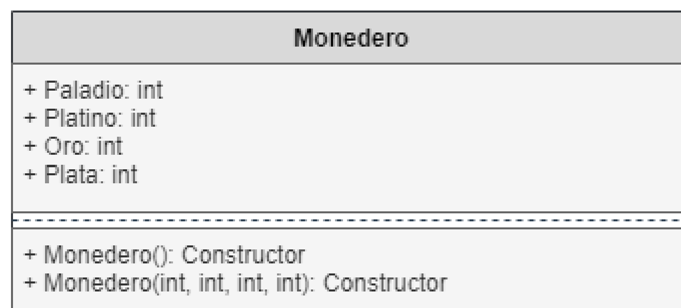

Figura 6.1.9/1: Repesentación de la clase Monedero.

El uso de esta clase se limita a definir los diferentes tipos de monedas existentes para permitir a los jugadores poder administrar el dinero de forma ordenada.

Cada atributo de la clase simboliza un tipo de moneda diferente, estos son:

- Paladio.
- Platino.
- Oro.
- Plata.

Esta clase solo tiene constructores como funciones.

6.1.10 Arma

Esta clase representa un arma y almacena toda la información que lo conforma en el sistema de rol. Ver Figura 6.1.10/1.

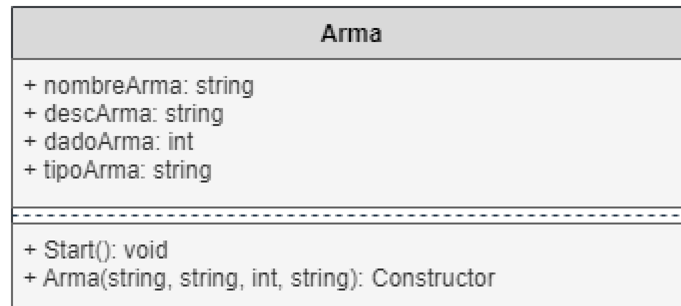

Figura 6.1.10/1: Repesentación de la clase Arma

El uso de esta clase se limita en definir la información que constituye un arma y que permite su uso correcto dentro de la aplicación.

Sus atributos son:

- **nombreArma:** El nombre del arma.
- **descArma:** Una corta descripción del arma.
- **dadoArma:** Representa el tipo de dado que se usa en las tiradas de ataque con el arma.
- **tipoArma:** Representa el valor al que está vinculado el arma y de los cuales se usa su bono para calcular el daño de sus ataques.

Su única función sin contar constructores se llama Start(). Esta función se usa en el primer frame en el que se activa la clase en la escena, y se limita a inicializar sus atributos.

6.2. Implementación del sistema de carga, guardado y borrado de personajes

Para cumplir con el objetivo que la aplicación se pueda usar a lo largo de varias partidas diferentes sin perder el progreso realizado y permitir a los usuarios usar hasta 4 personajes diferentes, es necesario implementar un sistema que guarde los datos de cada personaje de forma individual y que pueda cargar esos personajes en diferentes sesiones de uso. De forma complementaria, también es necesario implementar un sistema de borrado de esos datos por si el usuario desea eliminar personajes.

La solución encontrada ha sido usar un Singleton con el que la aplicación haga referencia en todo momento a cuál personaje debe consultar y manipular su información, y el uso de las clásicas ranuras de guardado para separar a los personajes y la carga de datos de cada uno de ellos.

6.2.1. Singleton

En la aplicación, el Singleton es la clase "DataPersistanceManager", explicada anteriormente. Esta clase sólo puede tener una instancia en cada escena, y cuando esta cambia, en vez de destruirse como el resto de elementos de una escena en Unity, se mantiene. Esto permite a la aplicación una forma de selección de personajes entre escenas. Ver **Figura 6.2.1/1**.

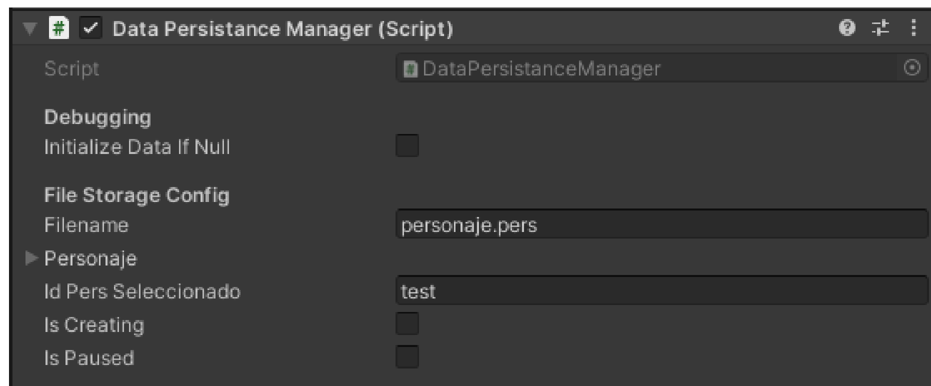

Figura 6.2.1/1: Imágen del Singleton en el inspector de Unity.

El gestor de personajes y el creador de personajes se encuentran en escenas diferentes al selector de personajes, así que, por cómo funcionan las escenas en Unity, es necesario un Singleton que mantenga la instancia del personaje seleccionado entre escenas para que la aplicación aplique los cambios correctamente. Este Singleton, en todas las escenas, se encuentra dentro de un asset llamado Personaje, compuesto sólo por un objeto simple con el script del Singleton. Ver **Figura 6.2.1/2**.

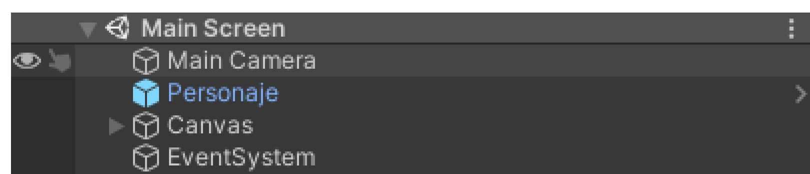

Figura 6.2.1/2: Imágen del Singleton "Personaje" en la escena del menú inicial, el selector de personajes.

6.2.2. Carga, guardado y borrado de personajes

Para que la aplicación pueda manipular la información de cada personaje de forma correcta, el Singleton siempre hace referencia a un personaje en concreto, que es seleccionado en el menú inicial, y utiliza el script "FileDataHandler" y sus funciones para lograr ese guardado, cargado y borrado de datos.

Al iniciarse, el Singleton crea una instancia del script "FileDataHandler" para poder usar sus funciones en su método "Awake". Para eso, le pasa la ruta del directorio en el que se encuentran los archivos de guardado de cada personaje con la función de Unity "Application.persistentDataPath", y el nombre del archivo de guardado que usarán sus funciones. Ver **Figura 6.2.2/1**.

```
private void Awake()
{
    if (instance != null)
    {
        Debug.Log("ERROR. Se ha encontrado mas de una instancia en la escena. Destruyendo la nueva...");
        Destroy(this.gameObject);
        return;
    }
    instance = this;
    DontDestroyOnLoad(this.gameObject);

    this.dataHandler = new FileDataHandler(Application.persistentDataPath, filename);
}
```

Figura 6.2.2/1: Imágen de la línea de código donde se genera el "FileDataHandler" en el Singleton.

Cuando el Singleton hace referencia a un personaje, siempre tiene guardado el código identificativo de ese personaje en su atributo "idPersSeleccionado". Esta es la manera que tiene el Singleton de comunicar al "FileDataHandler" de cuál personaje está pidiendo la manipulación de datos.

El cargado de los personajes puede ocurrir en varios momentos. Estos pueden ser:

- **Cargar una escena nueva:** Junto con la escena, el Singleton pide al "dataHandler" que cargue el personaje actual para mostrar siempre los datos guardados en el archivo.
- **Al actualizar un personaje:** Después de guardar los datos del personaje, los carga de nuevo para asegurar

que muestra los datos correctos almacenados en el sistema.

- **Al seleccionar un personaje:** Cuando se selecciona un personaje y teniendo su id, solicita al “dataHandler” cargar sus datos para poder mostrar la información del personaje seleccionado.
- **Volver de una pausa:** Cuando la aplicación vuelve a estar en primer plano, el Singleton solicita cargar de nuevo su información para asegurarse tener los datos actualizados del personaje que se habían guardado justo antes de pausar la aplicación.

Todas estas funciones, junto con el código de la función de cargado de personaje del Singleton, pueden observarse en el listado del código en el punto 6.1.2, así que no se volverán a mostrar en esta sección. Lo que tienen en común todos estos sistemas es que usan la función del Singleton “CargarPersonaje”, que a la vez, utilizan la función del “dataHandler” llamada “Cargar” para obtener cada personaje. Este es su código:

```
public Personaje Cargar(string idPers)
{
    string fullPath = Path.Combine(dataDirPath, idPers, dataFileName);
    Personaje dataCargado = null;
    if (File.Exists(fullPath))
    {
        try
        {
            string dataCargar = "";
            using (FileStream stream = new FileStream(fullPath,
                                                      FileMode.Open))
            {
                using (StreamReader reader = new StreamReader(stream))
                {
                    dataCargar = reader.ReadToEnd();
                }
            }

            dataCargado = JsonUtility.FromJson<Personaje>(dataCargar);
            Debug.Log("Personaje " + dataCargado.nombre + " cargado");
        }
        catch(Exception e)
        {
            Debug.LogError("ha ocurrido un error cargando el personaje desde
                            el archivo de guardado: " + fullPath + "\n" +
```

```
                                e);
            }
        }
        return dataCargado;
    }
```

Esta función devuelve un Personaje con los datos que haya encontrado en el archivo de guardado que coincida con el "idPersSeleccionado" del Singleton que la ha llamado y le ha pasado como parámetro. Este parámetro en esta función se llama "idPers", y se usa para que la función encuentre la ruta del archivo de guardado completo usando la función de Unity "Path.Combine". Esa ruta está conformada de la siguiente forma:

- **dataDirPath:** El directorio del sistema donde la aplicación guarda datos, atributo del script que se asigna por el Singleton al momento de crear su instancia.
- **idPers:** El código de identificación del personaje, que es el parámetro de la función. También simboliza un directorio, en este caso. Cada personaje siempre se guarda en un directorio único, con su respectivo código de identificación como nombre del directorio.
- **dataFileName**: El nombre que recibe el archivo de guardado, asignado también por el Singleton al momento de crear la instancia del "dataHandler".

Con la ruta del archivo encontrada, entonces usa un FileStream y un StreamReader, clases propias de Unity, para guardar todos los datos del archivo de guardado en un string llamado "dataCargar". Finalmente, asigna toda esa información a la variable de tipo Personaje llamada "dataCargado" usando el serializador del formato JSON en Unity, que es el formato en el que se guardan los datos en el archivo de guardado.

El guardado de personajes ocurre en estas ocasiones:

- **Antes de cambiar de escenas:** Para evitar que los cambios no se pierdan antes de que destruir todos los objetos de la escena, el Singleton se asegura de guardarlos.
- **Antes y después de pausar la aplicación:** Se guarda antes para tener al personaje actualizado por si luego se

cierra la aplicación, y después de volver solo si se está en el proceso de creación del personaje.

- **Al salir de la aplicación:** Para guardar todos los cambios hechos y que el personaje esté actualizado al iniciar la aplicación de nuevo.
- **Al actualizar el personaje**: Para poder actualizar los datos del personaje, primero se deben guardar en su respectivo archivo de guardado.

Todas estas funciones, junto con el código de la función de guardado de personaje del Singleton, pueden observarse en el listado del código en el punto 6.1.2, así que no se volverán a mostrar en esta sección. Lo que tienen en común todos estos momentos es que usan la función del Singleton "GuardarPersonaje", que a la vez, utilizan la función del "dataHandler" llamada "Guardar" para guardar cada personaje. Este es su código:

```
public void Guardar(Personaje pers, string idPers)
{
    string fullPath = Path.Combine(dataDirPath, idPers, dataFileName);
    Debug.Log(fullPath);
    try
    {
        Directory.CreateDirectory(Path.GetDirectoryName(fullPath));
        string dataGuardar = JsonUtility.ToJson(pers, true);

        using (FileStream stream = new FileStream(fullPath,
                                                  FileMode.Create))
        {
            using(StreamWriter writer = new StreamWriter(stream))
            {
                writer.Write(dataGuardar);
            }
        }
        Debug.Log("Personaje " + pers.nombre + " guardado");
    }
    catch (Exception e)
    {
        Debug.LogError("Ha ocurrido un error guardando el personaje en el
                        archivo de guardado: " + fullPath + "\n" + e);
    }
}
```

Esta función almacena todos los datos del Personaje entrado como parámetro en su respectivo archivo de guardado,

encontrado gracias al identificador del parámetro “idPers” de la función, pasado por el Singleton. Como en la función anterior, se genera la ruta del archivo de guardado de la misma forma, con la función “Path.Combine” y los parámetros “dataDirPath”, “idPers” y “dataFileName”. Luego crea el directorio único donde se guarda el archivo de guardado del personaje con la función del propio C# “Directory.CreateDirectory”. Esta función si encuentra que el archivo ya existe, simplemente no hace nada. Después crea un string llamado “dataGuardar” donde serializa los datos del Personaje en formato JSON con la función de Unity “JsonUtility.ToJson” y finalmente usa las funciones “FileStream” y “StreamWriter” para terminar escribiendo los datos de “dataGuardar” en el archivo de guardado.

El borrado de personajes ocurre en estas ocasiones:

- **Al dejar la aplicación en segundo plano:** En caso de estar en el proceso de creación de un personaje y en previsión de que el usuario puede cerrar la aplicación, como ese personaje está incompleto, no es necesario guardarlo, por lo tanto se eliminan todos sus datos.
- **Al cerrar la aplicación:** Por la misma razón a la anterior.
- **Cuando el usuario pulsa el botón de confirmación de borrado:** La única forma que tiene el usuario de borrar un personaje creado es confirmando el borrado a través de la pulsación de un botón.

Las dos primeras ocasiones al igual que la función de borrado principal pueden observarse en el listado del código de la clase “DataPersistanceManager” en el punto 6.1.2, así que no se volverán a mostrar en esta sección. La última ocasión se mostrará en el punto 6.3.2. Lo que tienen en común todos estos momentos es que usan la función del Singleton “EliminarPersonaje”, que a la vez, utilizan la función del “dataHandler” llamada “Eliminar” para borrar cada personaje. Este es su código:

```
public void Eliminar(string idPers)
    {
        if(idPers == null)
        {
            return;
        }

        string fullpath = Path.Combine(dataDirPath, idPers, dataFileName);
        try
        {
            if (File.Exists(fullpath))
            {
                Directory.Delete(Path.GetDirectoryName(fullpath), true);
                Debug.Log("Personaje " + idPers + " eliminado");
            }
            else
            {
                Debug.LogWarning("Error al eliminar el personaje en el
                                  directorio: " + fullpath);
            }
        }
        catch(Exception e)
        {
            Debug.LogError("Error al eliminar el personaje con ID: " + idPers +
                           " en el directorio " + fullpath + "\n" + e);
        }
    }
```

Eliminar a un Personaje simplemente significa eliminar su respectivo archivo de guardado, por lo tanto, el funcionamiento de esta función es sencillo. Primero, al igual que las funciones explicadas anteriormente, genera la ruta del fichero a eliminar usando la función "Path.Combine" y los parámetros "dataDirPath", "idPers" y "dataFileName", siendo "idPers" el parámetro que identifica el Personaje, y por ende el archivo de guardado específico que el Singleton ha solicitado eliminar. Despues de comprobar que ese archivo existe mediante el uso de la función de C# "File.Exists", procede a su eliminación usando la función de C# "Directory.Delete".

6.3. Implementación del menú inicial

El menú inicial es la primera pantalla con la que el jugador se encuentra al entrar a la aplicación y contiene el selector de personajes.

Desde ahí el jugador puede decidir si entrar al creador de personajes o al gestor de un personaje en específico.

En este menú, todos los elementos que lo conforman están en un *canvas*. Un *canvas* es un elemento usado por el motor Unity para mostrar elementos correspondientes a la interfaz del juego como pueden ser botones, imágenes o texto. En el caso del *canvas* de este menú, su propiedad "Canvas Scaler" se ha modificado respecto a sus valores por defecto. El modo de escalado de la interfaz está en "Scale with screen size", con una resolución de referencia de 800X600 píxeles y el modo de coincidencia de pantalla a "Match Width or Height", haciendo que el escalado de la pantalla coincida con la resolución de referencia por amplitud. Esto permite que todos los elementos del menú escalen en todas las pantallas en el ancho de forma perfecta. Eso también conlleva que no haya escalado en lo alto, y por lo tanto, la altura de todos los elementos se mantenga en todas las pantallas, sin excepción. Ver **Figura 6.3/1**.

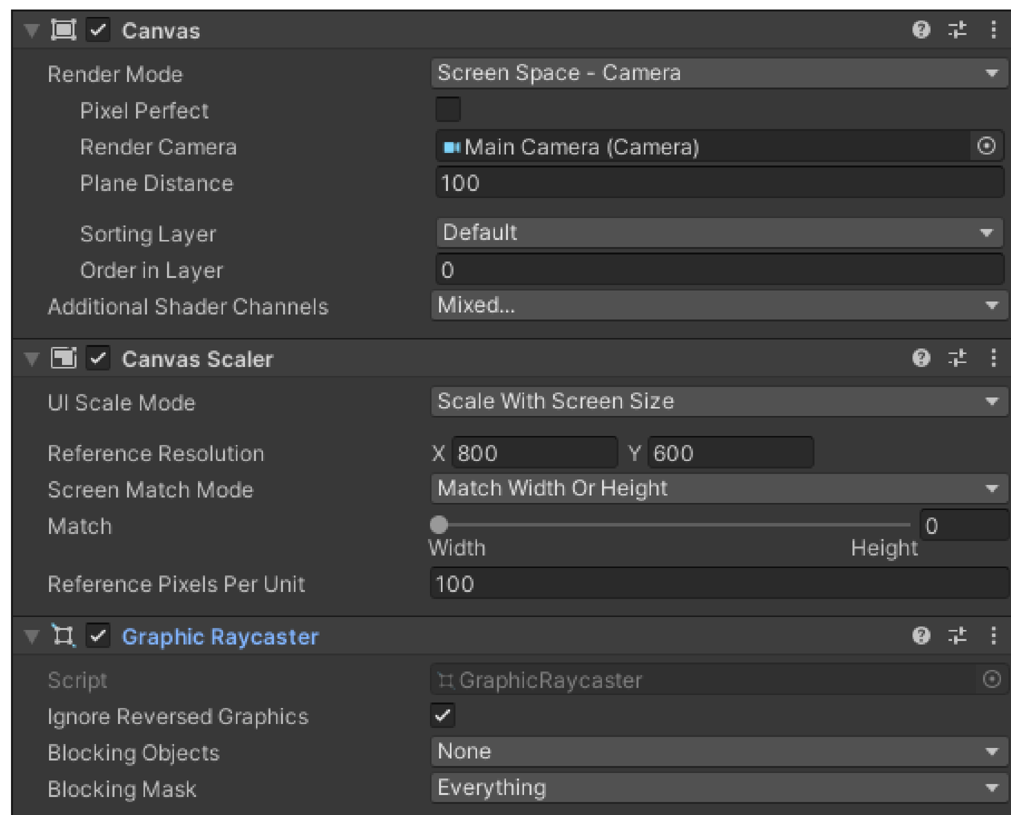

Figura 6.3/1: Imágen de los atributos del *canvas* en el inspector de Unity.

El orden de dibujado de cada elemento sigue el orden de la jerarquía, lo que permite esconder elementos "detrás" de otros elementos. Este menú está implementado aprovechando eso de tal forma que todos

los elementos que se pueden mostrar siempre están en el canvas, sacrificando tiempo de carga inicial a cambio de una carga inferior a durante la ejecución. Los elementos visibles son los que están al final de la jerarquía, y si hay que mostrar otros elementos que ocupan sus mismas posiciones, se activan o desactivan mediante la ejecución de código a tiempo real. Ver **Figura 6.3/2**.

Figura 6.3/2: Imágen de la jerarquía de elementos del *canvas* en Unity.

Para crear cada elemento que conforma no solo el menú inicial, sinó todas las partes de la aplicación, se han usado imágenes, botones, texto e *InputFields*.

6.3.1. Selector de personajes

Este selector funciona como los clásicos selectores de partidas guardadas de los videojuegos, donde cada una de las opciones representa a un personaje. A estas opciones, que hay 4, se le pueden llamar "ranuras" o "slots". Si en una ranura el personaje no existe, entonces se sustituye por un botón que lleva al jugador al creador de personajes. Si ese personaje existe, entonces mostrará información general del personaje que le corresponde, y al ser pulsada, llevará al jugador al gestor de ese personaje. Ver **Figura 6.3.1/1**.

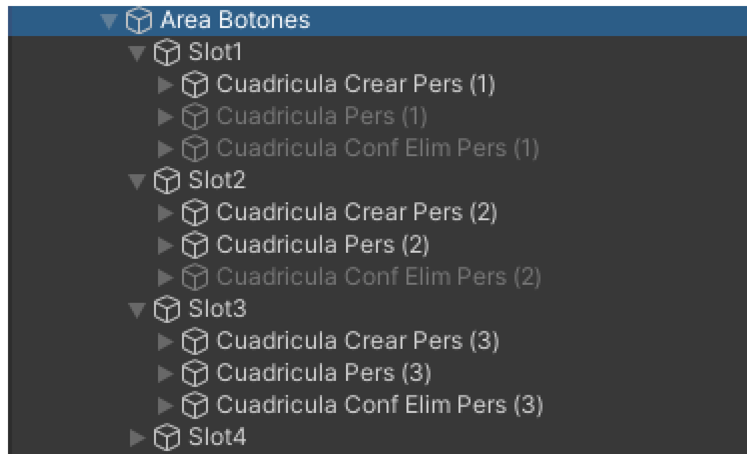

Figura 6.3.1/1: Imágen de la jerarquía de los slots en Unity.

En la **Figura 6.3.1/1** se puede observar la jerarquía de las ranuras de personaje que muestran en orden descendente el botón de crear personaje, el botón de personaje con su información y el mensaje de confirmación de borrado. Esta es la forma en la que la aplicación aprovecha el orden de dibujo del *canvas* de Unity para mostrar y dejar de mostrar elementos en pantalla comentada anteriormente.

El selector está compuesto por los 4 slots, y estos están reunidos en el objeto "Area Botones". Cada slot está compuesto por 3 elementos principales nombrados anteriormente:

- El botón de crear personaje, llamado en el motor "Cuadrícula Crear Pers". Ver **Figura 6.3.1/2**.

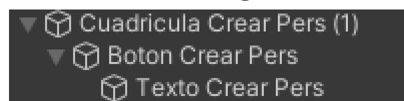

Figura 6.3.1/2: Imágen de la jerarquía del botón de crear personaje.

- El botón de personaje, llamado en el motor "Cuadrícula Pers". Ver **Figura 6.3.1/3.**

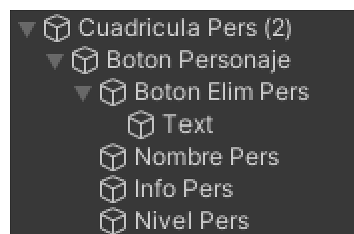

Figura 6.3.1/3: Imágen de la jerarquía del botón de personaje.

- El mensaje de confirmación de borrado, llamado en el motor "Cuadricula Conf Elim Pers". Ver **Figura 6.3.1/4.**

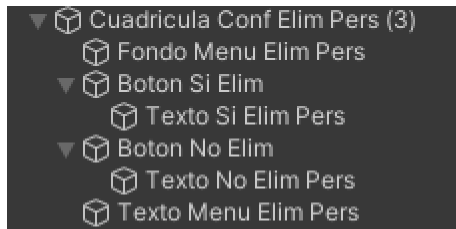

Figura 6.3.1/4: Imágen de la jerarquía del mensaje de confirmación de borrado.

Además, para controlar el comportamiento de cada slot, contienen un script llamado "Slot". Ver **Figura 6.3.1/5**.

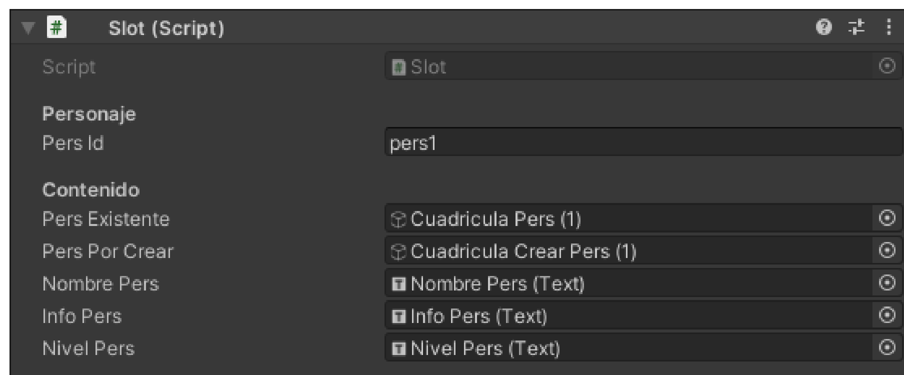

Figura 6.3.1/5: Imágen del script del primer slot en el inspector de Unity.

Sus variables son las siguientes:

- **persId:** Código de identificación que recibe el personaje del slot. Este es el que luego se usa en el Singleton para identificar a cada personaje.
- **persExistente:** Vincula la "Cuadrícula Pers" del slot para poder manipular su visibilidad.
- **persPorCrear:** Vincula la "Cuadrícula Crear Pers" del slot para poder manipular su visibilidad.
- **nombrePers/infoPers/nivelPers:** Vincula los textos de la "Cuadrícula Pers" del slot para mostrar la información del Personaje correspondiente.

Su función principal, llamada "PonInfo", se dedica a manipular el slot para visualizar los elementos adecuados de ese slot. Primero comprueba si el Personaje pasado por parámetro está vacío o no. Si lo está, significa que no existe, y por lo tanto, tendrá que mostrar el botón de crear personaje. De lo contrario, significa que sí existe, así que le añade su información básica al botón de personaje y lo muestra.

```
public void PonInfo(Personaje pers)
{
    if(pers == null)
    {
        persPorCrear.SetActive(true);
        persExistente.SetActive(false);
    }
    else
    {
        nombrePers.text = pers.nombre;
        infoPers.text = pers._raza.nombreRaza + " | " +
                        pers._clase.nombreClase;
        nivelPers.text = "Nivel: " + pers.nivel;

        persExistente.SetActive(true);
        persPorCrear.SetActive(false);
    }
}
```

Este script contiene otra función, llamada “GetIdPers”. Esta función simplemente devuelve el código de identificación del personaje del slot.

```
public string GetIdPers()
{
    return this.persId;
}
```

El elemento padre, “Area botones”, contiene el script “MenuSlots”. Este script se encarga de activar el menú, cargando los personajes existentes para que cada script “Slot” pueda obtener la información de su respectivo personaje en el momento de iniciar la aplicación y así pueda controlar correctamente el comportamiento de su slot asociado. Su funcionamiento es este:

```
private Slot[] saveSlots;

private void Awake()
{
    saveSlots = this.GetComponentsInChildren<Slot>();

}
```

Este script guarda una lista de los slots como única variable. En el momento de activarse mediante la función “Awake”, guarda en esa lista el script “Slot” de cada elemento hijo. En el primer frame, casi a la vez que la función anterior, ejecuta su función “Start” para a la vez llamar a su propia función llamada “Activar Menu”.

```
private void Start()
{
    ActivarMenu();
}
```

“Activar Menu” carga cada perfil de personaje existente, usando la función del Singleton “CargarPerfilesPersonaje”, y lo guarda en el diccionario “perfilesPers”. Con cada slot guardado en “saveSlots”, intenta tomar su personaje correspondiente del diccionario “perfilesPers” a partir de la clave con el uso de la función del propio Slot “GetIdPers” y poner su respectiva información en él mediante el uso de la función del mismo slot “PonInfo”, pasándole por parámetro dicho Personaje.

```
public void ActivarMenu()
{
    Dictionary<string, Personaje> perfilesPers =
    DataPersistanceManager.instance.CargarPerfilesPersonaje();
    foreach(Slot saveslot in saveSlots)
    {
        Personaje personaje = null;
        perfilesPers.TryGetValue(saveslot.GetIdPers(), out personaje);
        saveslot.PonInfo(personaje);
    }
}
```

La función anteriormente mencionada del Singleton, “CargarPerfilesPersonaje”, simplemente llama la función de su “dataHandler” llamada “CargarPerfiles”. Esta devuelve un diccionario de Personajes con su clave el código de identificación correspondiente a su slot.

```
public Dictionary<string, Personaje> CargarPerfilesPersonaje()
{
    return dataHandler.CargarPerfiles();
}
```

La función “CargarPerfiles” busca todos los directorios donde se guardan los archivos de guardado de los personajes en el directorio donde se guardan esos directorios usando la función de C# “EnumerateDirectories”. Su ruta la conoce ya que la tiene guardada en su atributo “dataDirPath”.

```
    public Dictionary<string, Personaje> CargarPerfiles()
    {
        Dictionary<string, Personaje> diccionarioPerfil = new Dictionary<string,
                                                              Personaje>();
        IEnumerable<DirectoryInfo> dirInfos = new
            DirectoryInfo(dataDirPath).EnumerateDirectories();
```

Por cada directorio encontrado, toma el identificador del personaje, que es el nombre del propio directorio, y lo guarda en la variable “idPerfil”. Con esto puede acceder al archivo de guardado creando la ruta completa con “Path.Combine”.

```
        foreach (DirectoryInfo dirInfo in dirInfos)
        {
            string idPerfil = dirInfo.Name;
            string fullpath = Path.Combine(dataDirPath, idPerfil, dataFileName);
            if (!File.Exists(fullpath))
            {
                Debug.LogWarning("Saltando directorio cargando los perfiles
                                  porque no contiene info: " + idPerfil);
                continue;
            }
```

Teniendo acceso, usa la función “Cargar” para guardar el personaje en una variable llamada “persPerfil”, y si ese personaje no está vacío, lo añadirá junto su respectivo identificador de “idPerfil” en el diccionario que terminará devolviendo la función.

```
                Personaje persPerfil = Cargar(idPerfil);
            if(persPerfil != null)
            {
                diccionarioPerfil.Add(idPerfil, persPerfil);
            }
            else
            {
                Debug.LogError("Ocurrió un error intentando cargar el personaje
                                de id: " + idPerfil);
            }
        }

        return diccionarioPerfil;
    }
```

6.3.2. Funcionalidad de los botones

El funcionamiento de los botones lo controla otro script llamado “FuncBotonPers”. En este se vinculan todos los elementos que interactúan con el input del jugador y se dedica a controlar el resto de componentes del menú. Este script se encuentra en un objeto dentro de la jerarquía de la escena llamado “Funcionabilidad”. Ver **Figura 6.3.2/1**.

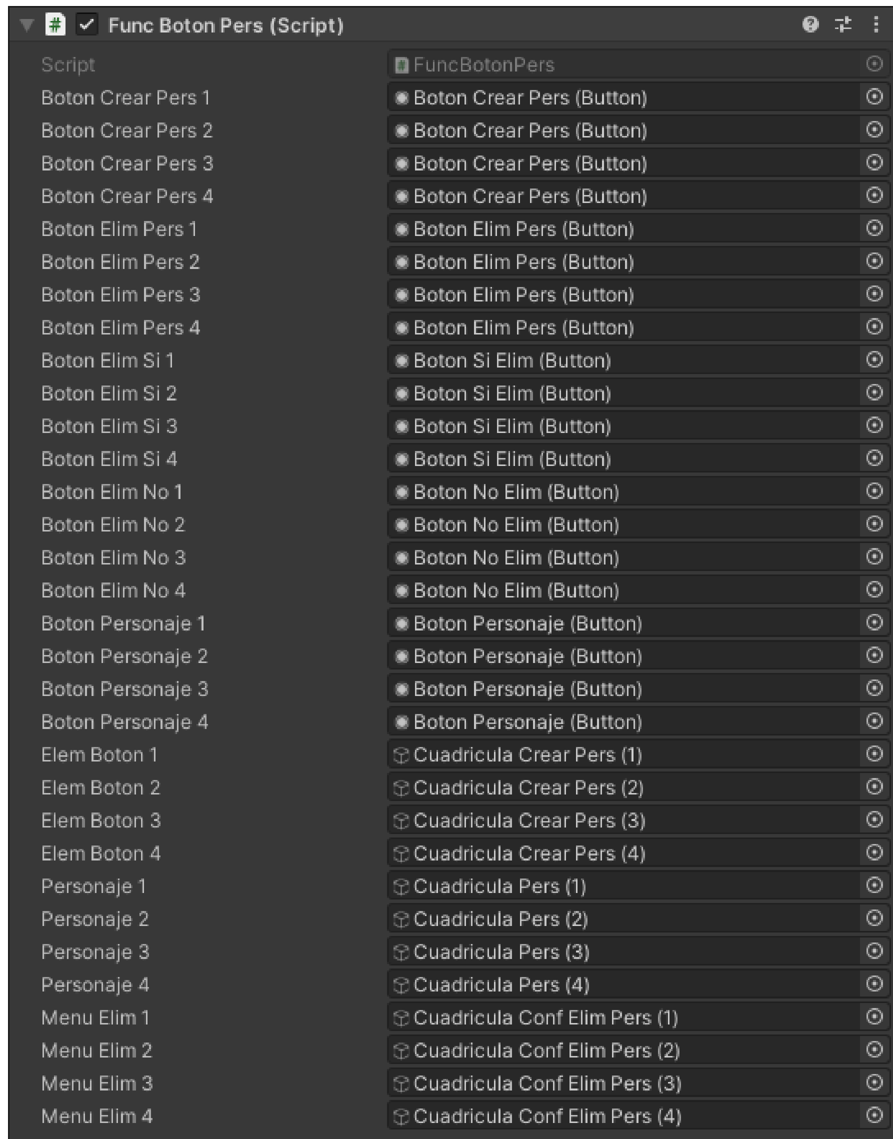

Figura 6.3.2/1: Imágen del script “FuncBotonPers” en el inspector de Unity.

Para empezar, usando la función “Start” se añade la funcionalidad de interacción de todos los botones del menú, a cada uno de ellos asignando una función llamada “Bypass” que llamará a la vez a otra función dependiendo del uso que tenga cada botón. Esto sirve para poder pasar como parámetros, variables personalizadas para cada botón.

```
void Start()
{
    botonCrearPers1.onClick.AddListener(Bypass1);
    botonCrearPers2.onClick.AddListener(Bypass2);
    botonCrearPers3.onClick.AddListener(Bypass3);
    botonCrearPers4.onClick.AddListener(Bypass4);

    botonElimPers1.onClick.AddListener(Bypass5);
    botonElimPers2.onClick.AddListener(Bypass6);
    botonElimPers3.onClick.AddListener(Bypass7);
    botonElimPers4.onClick.AddListener(Bypass8);

    botonElimSi1.onClick.AddListener(Bypass9);
    botonElimSi2.onClick.AddListener(Bypass10);
    botonElimSi3.onClick.AddListener(Bypass11);
    botonElimSi4.onClick.AddListener(Bypass12);

    botonElimNo1.onClick.AddListener(Bypass13);
    botonElimNo2.onClick.AddListener(Bypass14);
    botonElimNo3.onClick.AddListener(Bypass15);
    botonElimNo4.onClick.AddListener(Bypass16);

    botonPersonaje1.onClick.AddListener(CargarPersonaje1);
    botonPersonaje2.onClick.AddListener(CargarPersonaje2);
    botonPersonaje3.onClick.AddListener(CargarPersonaje3);
    botonPersonaje4.onClick.AddListener(CargarPersonaje4);

}
```

Hay 4 tipos de funciones "Bypass". La primera llama a la función "CrearPers" con los atributos personalizados de cada botón, que son los elementos del menú que cambian de estado al interactuar con ese botón, y el código de identificación del personaje.

```
void Bypass1()
{
    CrearPers(elemBoton1, personaje1, "pers1");
}
```

El segundo tipo llama a la función "MenuElimPers", que recibe como parámetros los elementos que cambian de estado al interactuar con el botón asociado.

```
void Bypass5()
{
    MenuElimPers(personaje1, menuElim1);
}
```

El tercer tipo de “Bypass” llama la función “ElimPers” con los parámetros a recibir siendo los elementos del menú que cambian de estado al interactuar con el botón asociado y el código de identificación del personaje.

```
void Bypass9()
{
    ElimPers(menuElim1, elemBoton1, "pers1");
}
```

Finalmente, el cuarto tipo llama a la función “NoElimPers”, que recibe como parámetros los elementos que cambian de estado al interactuar con el botón asociado.

```
void Bypass13()
{
    NoElimPers(menuElim1, personaje1);
}
```

La función “CrearPers” se usa cuando el botón de creación de personaje de un slot es pulsado, y se ocupa de que el Singleton cree ese personaje. Llama las funciones del Singleton “CambiarIdPersSelecc”, pasándole el identificador llamado “idPers” como parámetro, que es un parámetro que recibe esta misma función, “ChangeIsCreating” pasándole el valor true, ya que la aplicación pasará al creador de personajes, “CrearPersonaje” y “GuardarPersonaje”, y para completar el proceso, cambia de escena hacia la que corresponde del creador de personajes mediante la llamada de la función de Unity “SceneManager.LoadSceneAsync”.

```
void CrearPers(GameObject botonCrear, GameObject pers, string idPers)
{
    DataPersistanceManager.instance.CambiarIdPersSelecc(idPers);
    DataPersistanceManager.instance.ChangeIsCreating(true);
    DataPersistanceManager.instance.CrearPersonaje();
    DataPersistanceManager.instance.GuardarPersonaje();
    SceneManager.LoadSceneAsync("Creation");
}
```

La función "MenuElimPers" se ejecuta cuando el usuario pulsa el icono de papelera de basura, que inicia el proceso de eliminación del personaje al que corresponde el slot en el que se encuentra ese icono. Esta función se limita a mostrar el mensaje de confirmación usando la función de los objetos de Unity llamada "SetActive" con los parámetros *true* o *false* según lo que se necesite mostrar y ocultar; en este caso, mostrar el mensaje y ocultar el botón del personaje.

```
void MenuElimPers(GameObject pers, GameObject menuElim)
{
    pers.SetActive(false);
    menuElim.SetActive(true);
}
```

La función "ElimPers" inicia cuando el usuario confirma al pulsar el botón de "SÍ" del mensaje de confirmación para borrar un personaje. Entonces procede a la eliminación del personaje llamando la función del Singleton "EliminarPersonaje" y pasándole el identificador del personaje correspondiente al slot al que pertenece el botón. Luego oculta el mensaje de confirmación y activa el botón para crear un nuevo personaje en ese slot.

```
void ElimPers(GameObject menuElim, GameObject botonCrear, string idPers)
{
    DataPersistanceManager.instance.EliminarPersonaje(idPers);
    menuElim.SetActive(false);
    botonCrear.SetActive(true);
}
```

La función "NoElimPers" inicia cuando el usuario confirma al pulsar el botón de "NO" del mensaje de confirmación para borrar un personaje. Entonces, mediante el uso de la función

“SetActive”, oculta el mensaje y muestra de nuevo el botón del personaje del slot al que pertenece el botón pulsado.

```
    void NoElimPers(GameObject menuElim, GameObject pers)
    {
        menuElim.SetActive(false);
        pers.SetActive(true);
    }
```

Finalmente, existen 4 funciones más que tienen la misma funcionalidad llamadas “CargarPersonaje” seguido del número del personaje al que corresponden. Todas estas funciones se usan para acceder al gestor del personaje, pero cargando los datos de sus respectivos personajes. Al ejecutarse, llaman a las funciones del Singleton “CambiarIdPersSelecc”, pasándole el identificador perteneciente al personaje del botón para que cargue los datos del personaje seleccionado, y “GuardarPersonaje”. Finalmente, cambian la escena usando la acción de Unity “SceneManager.LoadSceneAsync”, dando paso al gestor del personaje.

```
void CargarPersonaje1()
    {
        DataPersistanceManager.instance.CambiarIdPersSelecc("pers1");
        DataPersistanceManager.instance.GuardarPersonaje();
        SceneManager.LoadSceneAsync("Character");
    }
```

6.4. Implementación del creador de personajes

El creador de personajes es donde el usuario confecciona a sus personajes. Para cumplir ese cometido, se ha generado una escena que contenga todas las mecánicas necesarias para que el usuario pueda crear un personaje que se pueda usar en partida.

Para implementar un creador fácil de entender para todos los usuarios, todos los elementos que se deben asignar a un personaje han sido organizados y separados en diferentes pasos en forma de tutorial. De esta forma, el usuario sólo debe de entender una parte del personaje a la vez en el momento de decidir sus elecciones.

Todos los scripts que controlan el comportamiento de todos los elementos del creador se encuentran en un objeto de la jerarquía llamado “Funcionabilidad”.

6.4.1. Navegación y organización

En esta escena todos los elementos que lo conforman están en un *canvas*. Ese canvas está implementado de la misma forma que el del menú inicial, explicado en el punto 6.3.

Para implementar esa separación de pasos en el creador de personajes, se han creado múltiples objetos que conforman las “pantallas” de la escena. Cada paso está formado por un cierto número de “pantallas”, dónde en cada uno de estos pasos se asigna un elemento esencial del personaje. En toda la escena hay un total de 23 “pantallas”, donde en cada una de ellas se muestran diferentes elementos. Ver **Figura 6.4.1/1**.

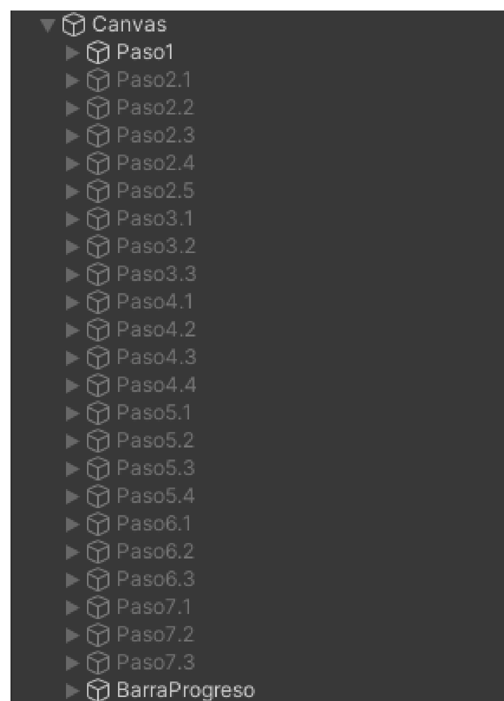

Figura 6.4.1/1: Captura de la jerarquía del *canvas* de la escena. Cada objeto a excepción de “BarraProgreso” representa una “pantalla”.

En el creador pueden haber dos tipos diferentes de “pantalla”. El primero, son las pantallas donde simplemente se dan explicaciones del elemento a asignar en el paso al que pertenecen y el proceso de asignación. El otro son las pantallas donde el usuario puede completar dicha asignación.

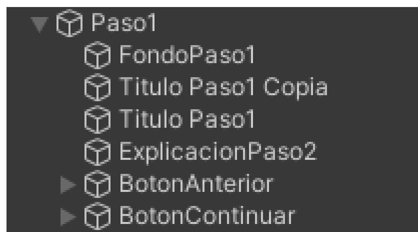

Figura 6.4.1/2: Captura de la jerarquía del paso 1 de la escena como ejemplo del primer tipo de pantalla.

Todas las pantallas del primer tipo comparten los mismos elementos vistos en la **Figura 6.4.1/2**.

Para que el usuario pueda navegar a través de las “pantallas” del creador de personajes, todas estas contienen 2 botones en la parte inferior de la pantalla con los que se puede avanzar a la siguiente pantalla o retroceder a la anterior. Esos botones son los que en la **Figura 6.4.1/2** tienen el nombre de “BotonContinuar” y “BotonAnterior” respectivamente.

El funcionamiento de todas las “pantallas” se encuentran en el objeto “Funcionamiento”, donde se guardan todos los scripts que controlan el comportamiento de todas las “pantallas”. Entre estos scripts se pueden diferenciar los que controlan las “pantallas” del primer tipo y las que controlan el segundo tipo. Las del primer tipo simplemente se limitan a añadir funcionalidad a los botones de avance y retroceso de cada pantalla, aunque para la última “pantalla”, también preparan varios elementos que se muestran en esas “pantallas” de asignación que las sigue. Cada script controla un paso en concreto, y todos los elementos asignados a cada script son los que manipula por código cada script. Ver **Figura 6.4.1/3**.

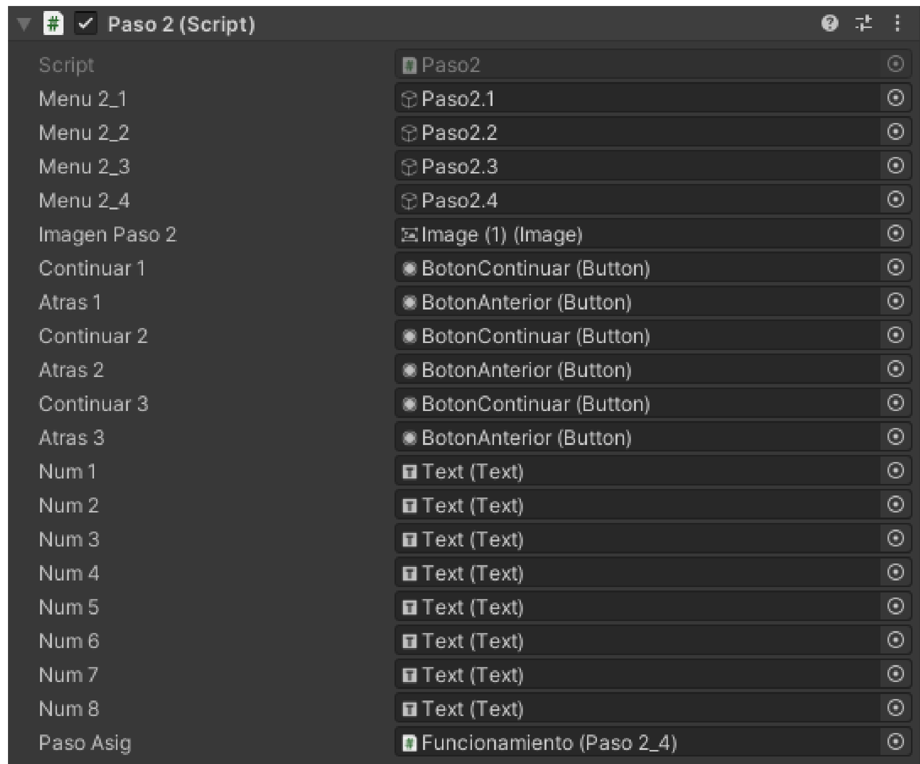

Figura 6.4.1/3: Captura del script en el inspector de Unity que controla las pantallas del primer tipo del paso 2, correspondiente a la asignación de valores.

Para mostrar el progreso del proceso de creación, existe una barra que muestra cuántos pasos se han completado en todo momento en forma de bloques, localizado en la parte superior de la pantalla. Esta barra se puede ver en la jerarquía de la **Figura 6.4.1/1** con el nombre de "BarraProgreso". Todos los scripts que controlan el comportamiento de avance o retroceso de "pantallas" cambian el color del bloque correspondiente al paso que muestran o dejan de mostrar. Si al avanzar una pantalla se termina un paso, el script se encarga de cambiar el color del bloque correspondiente en la barra de progreso al color verde, pero si se retrocede, se devuelve ese bloque al color blanco original.

El script de la **Figura 6.4.1/3**, similar al resto de scripts que controlan las "pantallas" de las explicaciones, siguen estas funciones:

```
void Start()
{
    continuar1.onClick.AddListener(Siguiente1);
    atras1.onClick.AddListener(Atras1);
    continuar2.onClick.AddListener(Siguiente2);
    atras2.onClick.AddListener(Atras2);
    continuar3.onClick.AddListener(Siguiente3);
    atras3.onClick.AddListener(Atras3);
}
```

En la función "Start" es donde se asignan las funciones que ejecutan los botones de avance y retroceso cuando son pulsados.

Las funciones con el nombre "Siguiente" sirven para activar la visibilidad de la siguiente "pantalla" mediante el uso de la función de Unity "IsActive" y pasando el parámetro *true*.

```
void Siguiente1()
{
    Menu2_2.SetActive(true);
}
```

En la última pantalla de todas, este tipo función lo que hace es terminar el proceso de creación de personaje, guardarlo y volver al menú inicial.

```
void Siguiente3()
{
    Personaje.nombre = nombreEntrado.text;
    DataPersistanceManager.instance.GuardarPersonaje();
    DataPersistanceManager.instance.ChangeIsCreating(false);
    SceneManager.LoadScene("Main Screen");
}
```

En caso de dar paso a una "pantalla" de asignación o selección, suelen llamarse más funciones que permiten preparar lo que esas pantallas deben mostrar. Un ejemplo de esto es la función "Siguiente3" de este mismo script.

```
void Siguiente3()
{
    CalcularNumeros();
    PasoAsig.ReiniciarPersonaje();
    Menu2_4.SetActive(true);
}
```

En este caso, en la función "CalcularNumeros" se hacen las tiradas para concretar los valores básicos que luego el usuario debe asignar a cada valor del personaje y en la función "ReiniciarPersonaje" se vuelven a poner los valores del personaje a sus valores por defecto para reiniciar el proceso de asignación. Estas funciones ya se mostrarán con más profundidad en el punto 6.4.2. Finalmente, simplemente muestra la pantalla de asignación.

Hay que tener en cuenta que todos los scripts que controlan este tipo de pantallas contienen este tipo de funciones para sus respectivos pasos. Estas se comentarán en sus respectivos apartados más adelante.

Finalmente están las funciones de nombre "Atras", con las que retroceden a la "pantalla" anterior simplemente quitando la visibilidad de la "pantalla" a la que pertenece el botón pulsado.

```
void Atras1()
{
    Menu2_1.SetActive(false);
    imagenPaso2.color = new Vector4(255, 255, 255, 100);
}
```

En este caso mostrado, como vuelve a un paso anterior, también le cambia el color del bloque correspondiente en la barra de progreso de la escena.

En la primera "pantalla", este tipo de función lo que hace es eliminar el personaje que se estaba creando porque no se ha cancelado el proceso de creación, terminar el proceso y cargar la escena del menú inicial.

```
void Atras()
{
    string persAct = DataPersistanceManager.instance.IdPersAct();
    DataPersistanceManager.instance.EliminarPersonaje(persAct);
    DataPersistanceManager.instance.ChangeIsCreating(false);
    SceneManager.LoadScene("Main Screen");
}
```

6.4.2. Asignación de valores

Corresponde a la finalización del paso 2 de la creación de un personaje, siendo la primera "pantalla" en la que el usuario debe tomar decisiones. En esta "pantalla", el usuario debe asignar unos números obtenidos a partir de las tiradas de los valores iniciales, explicadas en el punto 5.2.2.1 en los diferentes valores del personaje. Ver **Figura 6.4.2/1**.

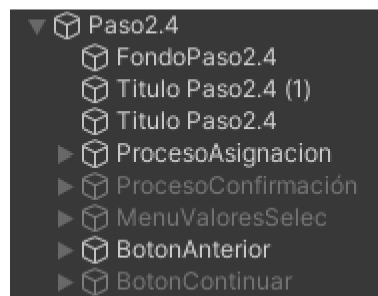

Figura 6.4.2/1: Captura de la jerarquía de la pantalla de asignación de valores.

Esta "pantalla" está dividida en dos partes que se muestran dependiendo de si la asignación está en proceso o ha terminado. Esta división la comparten todas las pantallas de asignación y selección. En este paso, esas dos partes se llaman "ProcesoAsignación" y "ProcesoConfirmación". Como se puede observar, el botón para seguir a la siguiente "pantalla" está bloqueado, y no se desbloquea hasta que el proceso haya terminado por completo.

En la primera parte, se muestran los resultados de dichas tiradas de dados. Cuando el usuario pulse uno de esos números, se mostrará el menú pop-up "MenuValoresSelec", donde el usuario puede seleccionar a qué valor asignar el número pulsado. Ver **Figura 6.4.2/2.**

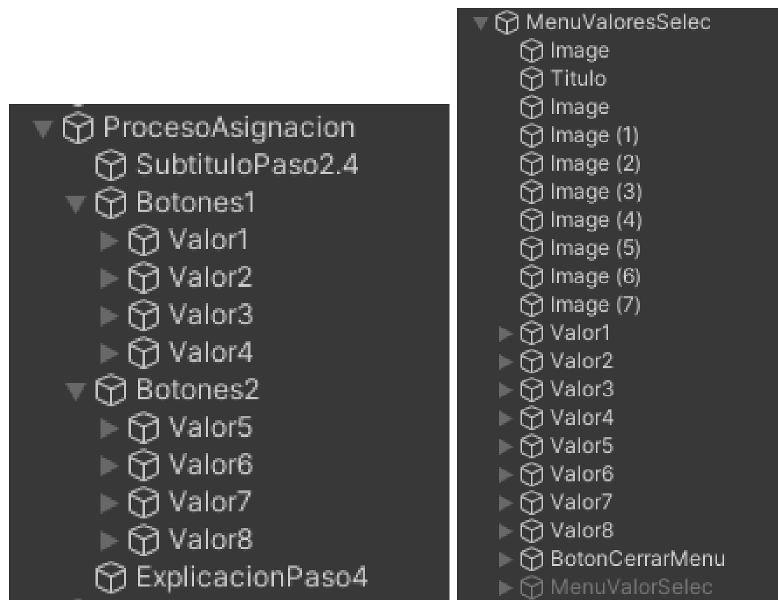

Figura 6.4.2/2: A la izquierda, captura de la jerarquía de la parte de asignación. A la derecha, captura de la jerarquía del menú de valores.

Cuando el usuario pulsa en uno de los valores, entonces se muestra el menú pop-up "MenuValorSelec", que sirve para confirmar la asignación del número al valor seleccionado. Ver **Figura 6.4.2/3**.

Figura 6.4.2/3: Captura de la jerarquía del menú de confirmación de asignación.

Cuando el usuario haya terminado de asignar todos los números en todos los valores, entonces se oculta la primera parte y se muestra la segunda parte. En esta, se muestra cada valor con el número asignado y se muestra una pregunta de confirmación de la asignación para el usuario. Ver **Figura 6.4.2/4**.

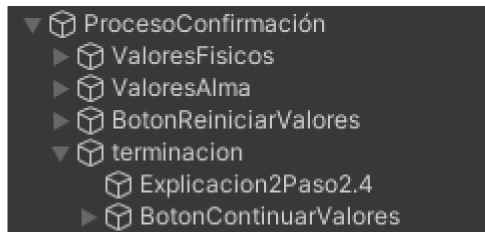

Figura 6.4.2/4: Captura de la jerarquía de la parte de confirmación.

El usuario tiene la opción de reiniciar la asignación, lo que volverá a mostrar la primera parte para empezar de nuevo, o la opción de confirmar, lo que desbloquea el botón para continuar a la siguiente pantalla. El botón de reinicio siempre está disponible incluso después de confirmar la asignación.

La funcionalidad de esta pantalla viene dada mayoritariamente por el script “Paso 2_4”, que controla casi todos los elementos de esta pantalla. Ver **Figura 6.4.2/5**.

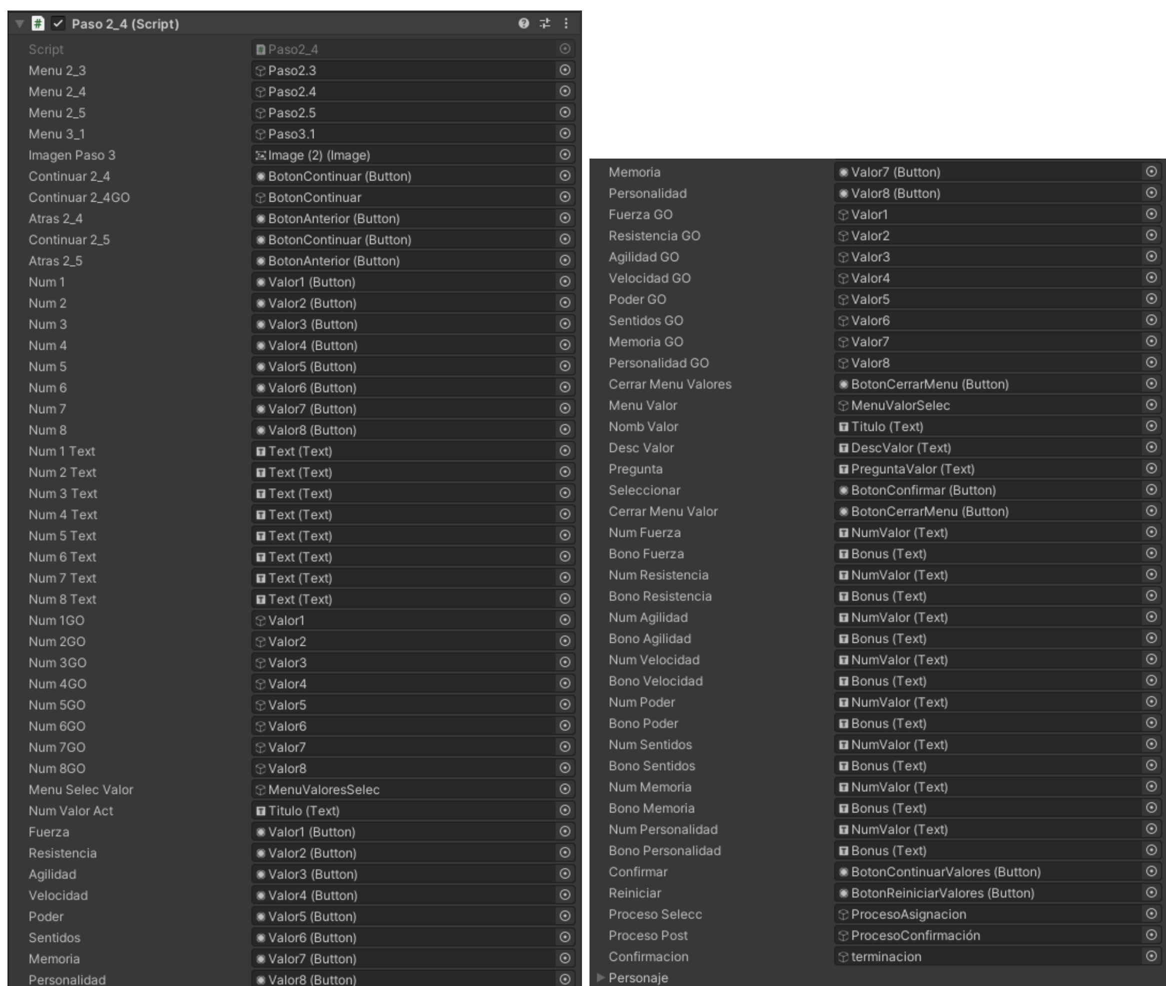

Figura 6.4.2/5: Capturas del script Paso 2_4 y sus atributos.

Los únicos elementos que no controla son los números resultantes de las tiradas de dados, que controla el script “Paso 2” con la función “CalcularNumeros”. Esta simula las tiradas de dados, teniendo en cuenta las normas que siguen en el sistema de rol para evitar crear personajes demasiado débiles. Cuando termina las tiradas, asigna los resultados a los números de la primera parte de la “pantalla”.

```
void CalcularNumeros()
    {
        int[] numeros = new int[8];
        int min = 4;
        int max = 17;
        int numMenores = 0;
        for (int i = 0; i < 8; i++)
        {
            int valor = Random.Range(min, max);

            while (valor < 6) { valor = Random.Range(min, max); }
            if (valor >= 6 && valor < 8) { numMenores++; }
            numeros[i] = valor;
        }
        int j = 0;
        while (numMenores >= 3)
        {
            if (numeros[j] < 8)
            {
                numeros[j] = Random.Range(8, max);
                numMenores--;
            }
            j++;
        }
        num1.text = numeros[0].ToString();
        num2.text = numeros[1].ToString();
        num3.text = numeros[2].ToString();
        num4.text = numeros[3].ToString();
        num5.text = numeros[4].ToString();
        num6.text = numeros[5].ToString();
        num7.text = numeros[6].ToString();
        num8.text = numeros[7].ToString();
    }
```

El script “Paso 2_4” contiene las siguientes funciones:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder manipular los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **Start():** Inicializa y asigna a todos los botones de la pantalla sus funciones para cuando son pulsados.

```
void Start()
{
    numValoresSelec = 0;
    continuar2_4.onClick.AddListener(Siguiente1);
    atras2_4.onClick.AddListener(Atras1);
    continuar2_5.onClick.AddListener(Siguiente2);
    atras2_5.onClick.AddListener(Atras2);
    num1.onClick.AddListener(Bypass1);
    num2.onClick.AddListener(Bypass2);
    num3.onClick.AddListener(Bypass3);
    num4.onClick.AddListener(Bypass4);
    num5.onClick.AddListener(Bypass5);
    num6.onClick.AddListener(Bypass6);
    num7.onClick.AddListener(Bypass7);
    num8.onClick.AddListener(Bypass8);
    fuerza.onClick.AddListener(Bypass9);
    resistencia.onClick.AddListener(Bypass10);
    agilidad.onClick.AddListener(Bypass11);
    velocidad.onClick.AddListener(Bypass12);
    poder.onClick.AddListener(Bypass13);
    sentidos.onClick.AddListener(Bypass14);
    memoria.onClick.AddListener(Bypass15);
    personalidad.onClick.AddListener(Bypass16);
    cerrarMenuValores.onClick.AddListener(CerrarPrimerMenu);
    cerrarMenuValor.onClick.AddListener(CerrarSegundoMenu);
    seleccionar.onClick.AddListener(AsignarValor);
    confirmar.onClick.AddListener(ConfirmarAsignacion);
    reiniciar.onClick.AddListener(ReiniciarAsignacion);
}
```

- **ReiniciarPersonaje():** Devolver los atributos del personaje a los valores por defecto. Usada cuando se reinicia el proceso de asignación.

```
public void ReiniciarPersonaje()
{
    Personaje.ActualizarValor("FUERZA", -Personaje.Valor("FUERZA"));
    Personaje.ActualizarValor("RESISTENCIA",
                              -Personaje.Valor("RESISTENCIA"));
    Personaje.ActualizarValor("AGILIDAD", -Personaje.Valor("AGILIDAD"));
    Personaje.ActualizarValor("VELOCIDAD", -Personaje.Valor("VELOCIDAD"));
    Personaje.ActualizarValor("PODER", -Personaje.Valor("PODER"));
    Personaje.ActualizarValor("SENTIDOS", -Personaje.Valor("SENTIDOS"));
    Personaje.ActualizarValor("MEMORIA", -Personaje.Valor("MEMORIA"));
    Personaje.ActualizarValor("PERSONALIDAD",
                              -Personaje.Valor("PERSONALIDAD"));
}
```

- **Bypass() 1-8:** Función para preparar y mostrar el menú pop-up de los valores donde se manipula el título para mostrar el número seleccionado.

```
void Bypass1()
{
    string numero = num1Text.text;
    numSelecBoton = num1GO;
    int.TryParse(numero, out num);
    numValorAct.text = "¿Cuál valor asignar " + numero + "?";
    MenuSelecValor.SetActive(true);
}
```

- **Bypass() 9-16:** Función para preparar y mostrar el menú pop-up de confirmación del valor seleccionado donde se manipulan varios elementos para mostrar la información correcta según las elecciones del usuario.

```
void Bypass9()
{
    string valor = "FUERZA";
    valorSelec = valor;
    nombValor.text = valor;
    descValor.text = "Simboliza cómo de fuerte es tu personaje. La habilidad
                      de manipular objetos pesados, golpear con potencia o
                      ejecutar acciones que requieran del uso de la fuerza
                      dependen de este valor.";
    pregunta.text = "¿Colocar " + num + " como valor base de " + valor +
                    "?";
    valorSelecBoton = fuerzaGO;
    MenuValor.SetActive(true);
}
```

- **AsignarValor():** Función usada cuando se asigna uno de los números a uno de los valores del personaje. También controla si se han asignado todos los valores para mostrar la segunda parte de la "pantalla".

```
void AsignarValor()
{
    Personaje.ActualizarValor(valorSelec, num);
    numValoresSelec++;
    numSelecBoton.SetActive(false);
    valorSelecBoton.SetActive(false);
    MenuValor.SetActive(false);
    MenuSelecValor.SetActive(false);
    if(numValoresSelec == 8)
    {
        ActValores();
        ProcesoSelecc.SetActive(false);
        ProcesoPost.SetActive(true);
    }
}
```

- **ActValores():** Función usada para actualizar los valores asignados al personaje.

```
public void ActValores()
{
    int numValor = Personaje.Valor("FUERZA");
    int bonoValor = Personaje.BonoValor("FUERZA");
    numFuerza.text = numValor.ToString();
    if (bonoValor >= 0) { bonoFuerza.text = "+" + bonoValor; }
    else { bonoFuerza.text = bonoValor.ToString(); }

    numValor = Personaje.Valor("RESISTENCIA");
    bonoValor = Personaje.BonoValor("RESISTENCIA");
    numResistencia.text = numValor.ToString();
    if (bonoValor >= 0) { bonoResistencia.text = "+" + bonoValor; }
    else { bonoResistencia.text = bonoValor.ToString(); }

    numValor = Personaje.Valor("AGILIDAD");
    bonoValor = Personaje.BonoValor("AGILIDAD");
    numAgilidad.text = numValor.ToString();
    if (bonoValor >= 0) { bonoAgilidad.text = "+" + bonoValor; }
    else { bonoAgilidad.text = bonoValor.ToString(); }

    numValor = Personaje.Valor("VELOCIDAD");
    bonoValor = Personaje.BonoValor("VELOCIDAD");
    numVelocidad.text = numValor.ToString();
    if (bonoValor >= 0) { bonoVelocidad.text = "+" + bonoValor; }
    else { bonoVelocidad.text = bonoValor.ToString(); }

    numValor = Personaje.Valor("PODER");
    bonoValor = Personaje.BonoValor("PODER");
    numPoder.text = numValor.ToString();
    if (bonoValor >= 0) { bonoPoder.text = "+" + bonoValor; }
    else { bonoPoder.text = bonoValor.ToString(); }

    numValor = Personaje.Valor("SENTIDOS");
    bonoValor = Personaje.BonoValor("SENTIDOS");
    numSentidos.text = numValor.ToString();
    if (bonoValor >= 0) { bonoSentidos.text = "+" + bonoValor; }
    else { bonoSentidos.text = bonoValor.ToString(); }

    numValor = Personaje.Valor("MEMORIA");
    bonoValor = Personaje.BonoValor("MEMORIA");
    numMemoria.text = numValor.ToString();
    if (bonoValor >= 0) { bonoMemoria.text = "+" + bonoValor; }
    else { bonoMemoria.text = bonoValor.ToString(); }

    numValor = Personaje.Valor("PERSONALIDAD");
    bonoValor = Personaje.BonoValor("PERSONALIDAD");
    numPersonalidad.text = numValor.ToString();
    if (bonoValor >= 0) { bonoPersonalidad.text = "+" + bonoValor; }
    else { bonoPersonalidad.text = bonoValor.ToString(); }
}
```

- **ConfirmarAsignacion():** Desbloquea el botón para continuar el proceso de creación.

```
void ConfirmarAsignacion()
{
    Confirmacion.SetActive(false);
    continuar2_4GO.SetActive(true);
}
```

- **ReiniciarAsignacion():** Devuelve todos los elementos gráficos de la pantalla al estado inicial y reinicia los valores del personaje llamando la función "ReiniciarPersonaje".

```
void ReiniciarAsignacion()
{
    num1GO.SetActive(true);
    num2GO.SetActive(true);
    num3GO.SetActive(true);
    num4GO.SetActive(true);
    num5GO.SetActive(true);
    num6GO.SetActive(true);
    num7GO.SetActive(true);
    num8GO.SetActive(true);
    numValoresSelec = 0;

    fuerzaGO.SetActive(true);
    resistenciaGO.SetActive(true);
    agilidadGO.SetActive(true);
    velocidadGO.SetActive(true);
    poderGO.SetActive(true);
    sentidosGO.SetActive(true);
    memoriaGO.SetActive(true);
    personalidadGO.SetActive(true);
    Confirmacion.SetActive(true);
    ProcesoPost.SetActive(false);
    ProcesoSelecc.SetActive(true);
    continuar2_4GO.SetActive(false);
    ReiniciarPersonaje();
}
```

- **Siguiente1():** Sirve para pasar a la siguiente pantalla. La última del paso 2.

```
void Siguiente1()
{
    Menu2_5.SetActive(true);
}
```

- **Siguiente2():** Sirve para pasar a la primera pantalla del tercer paso.

```
void Siguiente2()
{
    imagenPaso3.color = new Vector4(0, 255, 0, 100);
    Menu3_1.SetActive(true);
}
```

- **Atras1():** Sirve para volver a la pantalla anterior. Al hacer eso, también reinicia el proceso de asignación.

```
void Atras1()
{
    num1GO.SetActive(true);
    num2GO.SetActive(true);
    num3GO.SetActive(true);
    num4GO.SetActive(true);
    num5GO.SetActive(true);
    num6GO.SetActive(true);
    num7GO.SetActive(true);
    num8GO.SetActive(true);
    numValoresSelec = 0;

    fuerzaGO.SetActive(true);
    resistenciaGO.SetActive(true);
    agilidadGO.SetActive(true);
    velocidadGO.SetActive(true);
    poderGO.SetActive(true);
    sentidosGO.SetActive(true);
    memoriaGO.SetActive(true);
    personalidadGO.SetActive(true);

    continuar2_4GO.SetActive(false);
    MenuValor.SetActive(false);
    ProcesoSelecc.SetActive(true);
    Confirmacion.SetActive(true);
    ProcesoPost.SetActive(false);
    MenuSelecValor.SetActive(false);
    Menu2_4.SetActive(false);
    ReiniciarPersonaje();
```

```
    }
```

- **Atras2():** Sirve para volver a la pantalla de asignación de valores.

```
    void Atras2()
    {
        Menu2_5.SetActive(false);
    }
```

- **CerrarPrimerMenu():** Función usada para cerrar el menú pop-up “MenuValoresSelec”.

```
    void CerrarPrimerMenu()
    {
        MenuSelecValor.SetActive(false);
    }
```

- CerrarSegundoMenu(): Función usada para cerrar el menú pop-up “MenuValorSelec”.

```
    void CerrarSegundoMenu()
    {
        MenuValor.SetActive(false);
    }
```

6.4.3. Selección de raza

Corresponde a la finalización del paso 3 de la creación de un personaje. En esta “pantalla”, el usuario debe seleccionar la raza a la que quiere que su personaje pertenezca. Ver **Figura 6.4.3/1**.

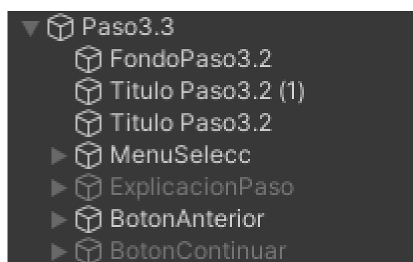

Figura 6.4.3/1: Captura de la jerarquía de la pantalla de selección de raza.

Esta “pantalla” está dividida en dos partes que se muestran dependiendo de si la selección está en proceso o ha terminado. En este paso, esas dos partes se llaman “MenuSelecc” y “ExplicacionPaso”. Como se puede observar, el botón para seguir a la siguiente “pantalla” está bloqueado, y no se desbloquea hasta que el proceso haya terminado por completo.

En la primera parte, se muestran las diferentes razas disponibles de Animaia en botones. Cuando el usuario pulse uno se esos botones, se mostrará el menú pop-up “MenuRaza”, donde el usuario podrá ver información de la raza seleccionada. Ver **Figura 6.4.3/2.**

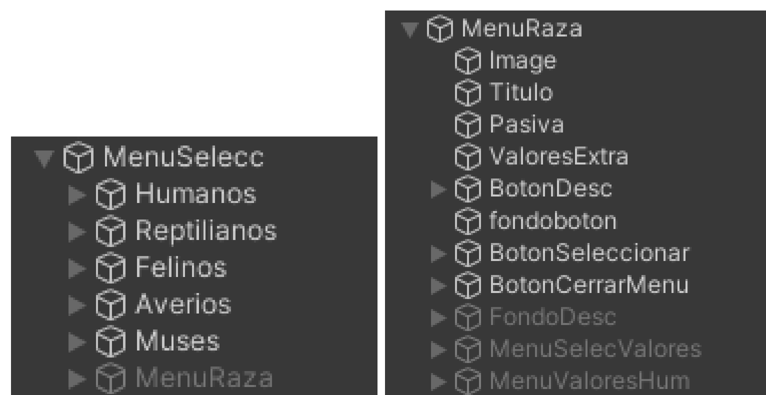

Figura 6.4.3/2: A la izquierda, captura de la jerarquía de la parte de las razas. A la derecha, captura de la jerarquía del menú de la raza seleccionada.

En este menú, el usuario puede confirmar la selección pulsando el botón “BotonSeleccionar”. Cuando se selecciona una raza, se inicia el proceso para que se seleccione el tercer valor vinculado a esa raza. Entonces, se activa el menú “MenuSelecValores”; en caso de que se seleccione la raza “Humano”, se activa el menú “MenuValoresHum” ya que sigue un proceso diferente. Ver **Figura 6.4.3/3.**

Figura 6.4.3/3: A la izquierda, captura del menú “MenuSelecValores”. A la derecha, captura de la jerarquía del menú “MenuValoresHum”.

En el menú “MenuSelecValores” el usuario sólo debe seleccionar uno de los dos valores que se muestran para asignarlo como el valor terciario de la raza, tal y como se explica en el manual de rol.

En el menú “MenuValoresHum”, por la naturaleza de la selección de la raza “Humano”, en la que se permite asignar cualquiera de los valores, se ha implementado de una forma diferente. Primero se muestra un menú donde se da a elegir si se quiere un valor físico o almático. Al pulsar una de esas opciones, se mostrarán sus respectivos valores. El primer valor seleccionado será el asignado como principal, que aumentará en 2 puntos y el segundo y tercero serán el secundario y terciario respectivamente, y aumentarán en 1 punto.

Cuando el usuario haya terminado de asignar los valores vinculados a la raza, entonces se oculta la primera parte y se muestra la segunda parte. En esta, se muestra un simple mensaje recordando la selección hecha. También se desbloquea el botón para continuar a la siguiente pantalla. Ver **Figura 6.4.3/4**.

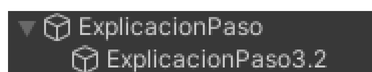

Figura 6.4.3/4: Captura de la jerarquía de la parte final.

La funcionalidad de esta pantalla proviene del mismo script que controla todo el tercer paso, llamado "Paso 3". Ver **Figura 6.4.3/5**.

Figura 6.4.3/5: Captura del script Paso 3 y sus atributos.

Este script contiene las siguientes funciones:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder manipular los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **Start():** Usada en el primer frame en el que se activa la escena. Llama a las funciones "IniciarBotones1", "IniciarBotones2", "IniciarBotones3" y "IniciarBotones4",

```
void Start()
{
    IniciarBotones1();
    IniciarBotones2();
    IniciarBotones3();
    IniciarBotones4();
}
```

- **IniciarBotones1():** Inicializa y asigna a todos los botones de avance y retroceso a sus funciones para cuando son pulsados.

```
void IniciarBotones1()
{
    continuar1.onClick.AddListener(Siguiente1);
    atras1.onClick.AddListener(Atras1);
    continuar2.onClick.AddListener(Siguiente2);
    atras2.onClick.AddListener(Atras2);
    continuar3.onClick.AddListener(Siguiente3);
    atras3.onClick.AddListener(Atras3);
}
```

- **IniciarBotones2():** Inicializa y asigna todos los botones de selección de raza sus funciones para cuando son pulsados.

```
void IniciarBotones2()
{
    humanos.onClick.AddListener(Bypass1);
    reptilianos.onClick.AddListener(Bypass2);
    felinos.onClick.AddListener(Bypass3);
    averios.onClick.AddListener(Bypass4);
    muses.onClick.AddListener(Bypass5);
}
```

- **IniciarBotones3():** Inicializa y asigna a todos los botones del menú de raza sus funciones para cuando son pulsados.

```
void IniciarBotones3()
{
    AbrirDesc.onClick.AddListener(AbrirDescripcion);
    SelecRaza.onClick.AddListener(RazaSeleccionada);
    CerrarMenuRaza.onClick.AddListener(CerrarRaza);
    CerrarMenuDescRaza.onClick.AddListener(CerrarDescRaza);
}
```

- **IniciarBotones4():** Inicializa y asigna a todos los botones de selección del valor terciario de la raza seleccionada sus funciones para cuando son pulsados.

```
void IniciarBotones4()
{
    botonValor1.onClick.AddListener(SelecValor1);
    botonValor2.onClick.AddListener(SelecValor2);
}
```

- **guardarValoresPers():** Función usada para preparar los elementos del menú de selección de los valores de la raza “Humano”. También guarda los valores del personaje previamente a que estos cambien al seleccionar su raza. Se activa al entrar a la “pantalla de selección”.

```
void guardarValoresPers()
{
    descPaso.text = "Selecciona cual Valor quieres añadir +2";
    valorFuerza = Personaje.Valor("FUERZA");
    valorResist = Personaje.Valor("RESISTENCIA");
    valorAgilid = Personaje.Valor("AGILIDAD");
    valorVeloci = Personaje.Valor("VELOCIDAD");
    valorPoder = Personaje.Valor("PODER");
    valorSentid = Personaje.Valor("SENTIDOS");
    valorMemoria = Personaje.Valor("MEMORIA");
    valorPerson = Personaje.Valor("PERSONALIDAD");
}
```

- **SelecValor() 1-2:** Funciones usadas cuando se asigna el valor terciario de una raza que no sea "Humano".

```
void SelecValor1()
{
    Personaje._raza.valorTerciario = nomValor1.text;
    Personaje.ActualizarValor(nomValor1.text, 1);
    SeleccionRaza();
}

void SelecValor2()
{
    Personaje._raza.valorTerciario = nomValor2.text;
    Personaje.ActualizarValor(nomValor2.text, 1);
    SeleccionRaza();
}
```

- **SeleccionRaza():** Usada cuando termina el proceso de selección de cualquier raza, prepara y muestra la segunda parte de la "pantalla" mientras que oculta la primera.

```
void SeleccionRaza()
{
    Personaje._raza.nombreRaza = NombreRaza.text;
    Personaje._raza.descRaza = DescRaza.text;
    CerrarRaza();
    menuRazas.SetActive(false);
    ExplicacionPaso.text = "Raza seleccionada:\n\n " + NombreRaza.text;
    MenuValoresSelecNormal.SetActive(false);
    ExplicacionPasoGO.SetActive(true);
    continuar3GO.SetActive(true);
}
```

- **ReiniciarProceso():** Esta función devuelve todos los elementos de la "pantalla" a sus estados por defecto y al personaje a como estaba antes de empezar el proceso de selección de raza. Se usa cuando el usuario retrocede a la "pantalla" anterior a la de selección.

```
void ReiniciarProceso()
{
    CerrarRaza();
    MenuValoresSelecNormal.SetActive(false);
    MenuRazaSelec.SetActive(false);
    menuRazas.SetActive(true);
    ExplicacionPasoGO.SetActive(false);
    continuar3GO.SetActive(false);
    Personaje.ActualizarValores(valorFuerza, valorResist, valorAgilid,
                                valorVeloci, valorPoder, valorSentid,
                                valorMemoria, valorPerson);
}
```

- **Bypass() 1-5:** Función ejecutada cuando se pulsa uno de los botones de las razas disponibles. Se ocupa de preparar y mostrar el menú de la raza seleccionada con la información pertinente.

```
void Bypass1()
{
    string nombreRaza = "Humano";
    string descRaza = "Los seres más comunes del planeta, y los más
                       versátiles. Pueden hacer de todo si se lo proponen.
                       Se dice que eso es posible debido a que el potencial
                       de la humanidad es infinito. Están esparcidos por
                       todos los rincones del mundo y gobiernan la gran
                       mayoría de la población. Su control sobre todas las
                       sociedades es casi absoluto gracias a la capacidad
                       que tienen para cooperar y negociar con el resto de
                       razas a lo largo de la historia. Cuando la política
                       no daba sus frutos, su gran capacidad de estratagemas
                       en combate y sus avances en casi cada campo
                       científico les aseguraba casi siempre la ventaja, así
                       como su dominio en el campo de batalla. Se relacionan
                       generalmente bien con todas las razas, viviendo en
                       armonía en los centros urbanos, pero hay pequeños
                       grupos que aún siguen teniendo rencor a otras razas
                       debido a las diferencias de cultura y guerras del
                       pasado.";
    string nombrePasiva = "Versatilidad";
    string descPasiva = "+2, +1 y +1 a 3 Valores diferentes.";
    string valoresOpc = "Cualquiera";
    AbrirMenuRaza(nombreRaza, descRaza, nombrePasiva, descPasiva,
                  valoresOpc);
}
```

- **AbrirMenuRaza():** Se ocupa de mostrar el menú de la raza seleccionada con la información pertinente.

```
void AbrirMenuRaza(string nombreRaza, string descRaza, string nombrePasiva,
                   string descPasiva, string valoresOpc)
{
    NombreRaza.text = nombreRaza;
    DescRaza.text = descRaza;
    NombrePasiva.text = "Habilidad Pasiva: " + nombrePasiva + "\n" +
                        descPasiva;
    ValoresOpcionales.text = "Valores opcionales:\n" + valoresOpc;
    MenuRazaSelec.SetActive(true);
}
```

- **AbrirDescripcion():** Esta función sólo muestra la descripción del personaje activando el objeto donde se encuentra.

```
void AbrirDescripcion()
{
    MenuDescRaza.SetActive(true);
}
```

- **RazaSeleccionada():** Función usada para asignar y actualizar los valores vinculados a la raza en el personaje. En esta parte, lo hace cuando se asigna la raza “Reptiliano”:

```
void RazaSeleccionada()
{
    int valor1 = 0;
    int valor2 = 0;
    int bono1 = 0;
    int bono2 = 0;
    int valorSumado1 = 0;
    int bonoSumado1 = 0;
    int valorSumado2 = 0;
    int bonoSumado2 = 0;
    //Abrir menú de seleccion de Valores segun la raza
      seleccionada(NombreMenu)
    if (NombreRaza.text == "Reptiliano")
    {
        Personaje.ActualizarValor("RESISTENCIA", 2);
        Personaje.ActualizarValor("FUERZA", 1);
        Personaje._raza.valorPrincipal = "RESISTENCIA";
        Personaje._raza.valorSecundario = "FUERZA";
        CalcularValores(valor1, valor2, bono1, bono2, valorSumado1,
                        valorSumado2, bonoSumado1, bonoSumado2, "AGILIDAD",
                        "PERSONALIDAD");
    }
```

Esta es la parte que se activa cuando se selecciona la raza “Felino”:

```
    else if (NombreRaza.text == "Felino")
    {
        Personaje.ActualizarValor("AGILIDAD", 2);
        Personaje.ActualizarValor("PERSONALIDAD", 1);
        Personaje._raza.valorPrincipal = "AGILIDAD";
        Personaje._raza.valorSecundario = "PERSONALIDAD";
        CalcularValores(valor1, valor2, bono1, bono2, valorSumado1,
                        valorSumado2, bonoSumado1, bonoSumado2, "FUERZA",
                        "VELOCIDAD");
    }
```

Esta es la parte que se activa cuando se selecciona la raza “Averio”:

```
        else if (NombreRaza.text == "Averio")
        {
            Personaje.ActualizarValor("SENTIDOS",  2);
            Personaje.ActualizarValor("VELOCIDAD", 1);
            Personaje._raza.valorPrincipal = "SENTIDOS";
            Personaje._raza.valorSecundario = "VELOCIDAD";
            CalcularValores(valor1, valor2, bono1, bono2, valorSumado1,
                            valorSumado2, bonoSumado1, bonoSumado2, "MEMORIA",
                            "PODER");
        }
```

Esta es la parte que se activa cuando se selecciona la raza “Mus”:

```
        else if (NombreRaza.text == "Mus")
        {
            Personaje.ActualizarValor("PODER", 2);
            Personaje.ActualizarValor("MEMORIA", 1);
            Personaje._raza.valorPrincipal = "PODER";
            Personaje._raza.valorSecundario = "MEMORIA";
            CalcularValores(valor1, valor2, bono1, bono2, valorSumado1,
                            valorSumado2, bonoSumado1, bonoSumado2,
                            "RESISTENCIA", "SENTIDOS");
        }
```

Finalmente, para la raza “Humano”, llama a la función “SelecValores” para poder preparar el menú “MenuValoresHum”.

```
        else if (NombreRaza.text == "Humano")
        {
            funcionHumano.SelecValores();
        }
    }
```

- **CalcularValores():** Función usada para calcular y actualizar los valores terciarios vinculados a la raza del personaje. Sirve para preparar el menú de selección del valor terciario de la raza.

```
void CalcularValores(int Val1, int Val2, int Bon1, int Bon2, int ValSum1, int
                     ValSum2, int BonoSum1, int BonoSum2, string Valor1, string
                     Valor2)
    {
        Val1 = Personaje.Valor(Valor1);
        Val2 = Personaje.Valor(Valor2);
        Bon1 = Personaje.BonoValor(Valor1);
        Bon2 = Personaje.BonoValor(Valor2);
        ValSum1 = Val1 + 1;
        ValSum2 = Val2 + 1;
        if (ValSum1 >= 8) { BonoSum1 = (ValSum1 - 8) / 2; }
        else { BonoSum1 = (ValSum1 - 9) / 2; }
        if (ValSum2 >= 8) { BonoSum2 = (ValSum2 - 8) / 2; }
        else { BonoSum2 = (ValSum2 - 9) / 2; }

        numeroValor1.text = Val1 + " > " + ValSum1;
        bonoValor1.text = Bon1 + " > " + BonoSum1;
        numeroValor2.text = Val2 + " > " + ValSum2;
        bonoValor2.text = Bon2 + " > " + BonoSum2;

        nomValor1.text = Valor1;
        nomValor2.text = Valor2;
        MenuValoresSelecNormal.SetActive(true);
    }
```

- **CerrarRaza():** Función usada para desactivar el menú pop-up de la raza.

```
    void CerrarRaza()
    {
        MenuRazaSelec.SetActive(false);
    }
```

- **CerrarDescRaza():** Función usada para desactivar el objeto donde se encuentra la descripción de la raza.

```
    void CerrarDescRaza()
    {
        MenuDescRaza.SetActive(false);
    }
```

Para vincular los valores a la raza cuando se selecciona la raza “Humano”, se usa la función “SelecValores”, que se encuentra en el script “SelValorHuman”. Ver **Figura 6.4.3/6.**

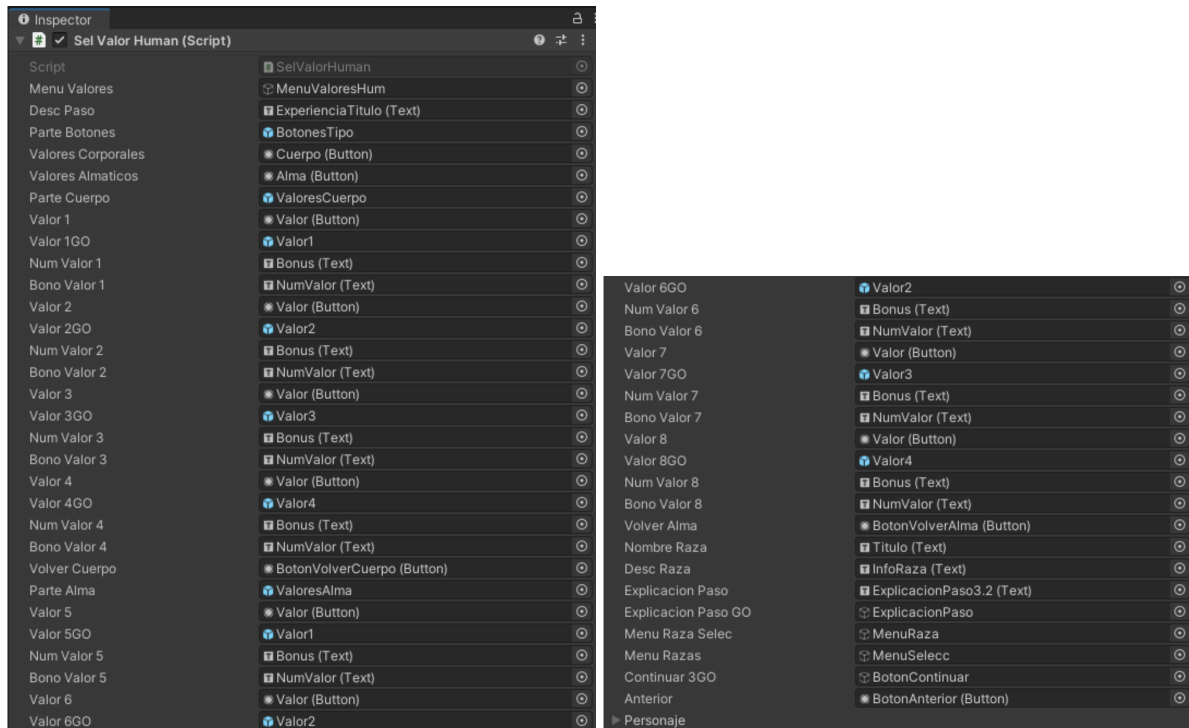

Figura 6.4.3/6: Capturas del script SelValorHuman y sus atributos.

Sus funciones son:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder manipular los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **Start():** Inicializa y asigna a todos los botones del menú de asignación de valores de raza sus funciones para cuando son pulsados.

```
void Start()
{
    ValoresCorporales.onClick.AddListener(AbrirValoresCuerpo);
    ValoresAlmaticos.onClick.AddListener(AbrirValoresAlma);
    IniciarBotonesValores();
    VolverCuerpo.onClick.AddListener(CerrarValoresCuerpo);
    VolverAlma.onClick.AddListener(CerrarValoresAlma);
    anterior.onClick.AddListener(Reiniciar);
}
```

- **Reiniciar():** Devuelve al estado por defecto todos los elementos del menú de selección de valores de la raza.

```
void Reiniciar()
{
    Valor1GO.SetActive(true);
    Valor2GO.SetActive(true);
    Valor3GO.SetActive(true);
    Valor4GO.SetActive(true);
    Valor5GO.SetActive(true);
    Valor6GO.SetActive(true);
    Valor7GO.SetActive(true);
    Valor8GO.SetActive(true);
    MenuValores.SetActive(false);
    ParteBotones.SetActive(true);
    ParteCuerpo.SetActive(false);
    ParteAlma.SetActive(false);
    numselec = 0;
}
```

- **AbrirValoresCuerpo():** Función usada para mostrar los valores corporales en el menú.

```
void AbrirValoresCuerpo()
{
    ParteBotones.SetActive(false);
    ParteCuerpo.SetActive(true);
}
```

- **AbrirValoresAlma():** Función usada para mostrar los valores almáticos en el menú.

```
void AbrirValoresAlma()
{
    ParteBotones.SetActive(false);
    ParteAlma.SetActive(true);
}
```

- **CerrarValoresCuerpo():** Función usada para ocultar los valores corporales y mostrar de nuevo los botones iniciales del menú.

```
void CerrarValoresCuerpo()
{
    ParteCuerpo.SetActive(false);
    ParteBotones.SetActive(true);
}
```

- **CerrarValoresAlma():** Función usada para ocultar los valores almáticos y mostrar de nuevo los botones iniciales del menú.

```
void CerrarValoresAlma()
{
    ParteAlma.SetActive(false);
    ParteBotones.SetActive(true);
}
```

- **IniciarBotonesValores():** Asigna a todos los botones relacionados con los valores del menú sus funciones para cuando son pulsados.

```
void IniciarBotonesValores()
{
    Valor1.onClick.AddListener(Bypass1);
    Valor2.onClick.AddListener(Bypass2);
    Valor3.onClick.AddListener(Bypass3);
    Valor4.onClick.AddListener(Bypass4);
    Valor5.onClick.AddListener(Bypass5);
    Valor6.onClick.AddListener(Bypass6);
    Valor7.onClick.AddListener(Bypass7);
    Valor8.onClick.AddListener(Bypass8);
}
```

- **Bypass() 1-8:** Llama la función "SeleccionarValor" pasándole por referencia los parámetros pertinentes del botón pulsado.

```
    void Bypass1()
    {
        SeleccionarValor("FUERZA", Valor1GO);
    }
```

- **SeleccionarValor():** Función usada para asignar cada valor seleccionado a la raza del personaje, actualizando dicho valor adecuadamente. Cuando se selecciona un valor, deja de estar disponible desactivando su visibilidad.

```
void SeleccionarValor(string valor, GameObject objetoValor)
    {
        if (numselec == 0)
        {
            Personaje._raza.valorPrincipal = valor;
            Personaje.ActualizarValor(valor, 2);
            descPaso.text = "Selecciona cual Valor quieres añadir +1";
            CalcularValoresSelec();
        }
        else if (numselec == 1)
        {
            Personaje._raza.valorSecundario = valor;
            Personaje.ActualizarValor(valor, 1);
        }
        else
        {
            Personaje._raza.valorTerciario = valor;
            Personaje.ActualizarValor(valor, 1);
            SeleccionRaza();
        }
        objetoValor.SetActive(false);
        numselec++;
    }
```

- **SeleccionRaza():** Función usada cuando todos los valores han sido seleccionados, terminando el proceso de la primera parte y activando la segunda parte de la "pantalla".

```
void SeleccionRaza()
{
    CerrarValoresCuerpo();
    CerrarValoresAlma();
    Personaje._raza.nombreRaza = NombreRaza.text;
    Personaje._raza.descRaza = DescRaza.text;
    MenuRazaSelec.SetActive(false);
    menuRazas.SetActive(false);
    ExplicacionPaso.text = "Raza seleccionada:\n\n " + NombreRaza.text;
    MenuValores.SetActive(false);
    ExplicacionPasoGO.SetActive(true);
    continuar3GO.SetActive(true);
    numselec = 0;
}
```

- **SelecValores():** Función usada para inicializar y activar el menú de selección de los valores de la raza.

```
public void SelecValores()
{
    numselec = 0;
    CalcularValoresInicio();
    MenuValores.SetActive(true);
}
```

- **CalcularValoresInicio():** Función usada para calcular todos los valores a mostrar cuando inicia la selección el valores. El valor principal de la raza, que aumenta en 2 puntos, debe reflejarse en todos los valores para que el usuario sepa el resultado de su selección. Llama a la función "CalcularValores2" para que lo haga con cada valor.

```
void CalcularValoresInicio()
{
    CalcularValores2("FUERZA", numValor1, bonoValor1);
    CalcularValores2("RESISTENCIA", numValor2, bonoValor2);
    CalcularValores2("AGILIDAD", numValor3, bonoValor3);
    CalcularValores2("VELOCIDAD", numValor4, bonoValor4);
    CalcularValores2("PODER", numValor5, bonoValor5);
    CalcularValores2("SENTIDOS", numValor6, bonoValor6);
    CalcularValores2("MEMORIA", numValor7, bonoValor7);
    CalcularValores2("PERSONALIDAD", numValor8, bonoValor8);
}
```

- **CalcularValoresSelec():** Función usada para calcular todos los valores a mostrar cuando se selecciona el valor principal de la raza, ya que ahora solo se debe mostrar el cambio de cada valor por 1 punto. Llama a la función "CalcularValores1" para que lo haga con cada valor.

```
void CalcularValoresSelec()
{
    CalcularValores1("FUERZA", numValor1, bonoValor1);
    CalcularValores1("RESISTENCIA", numValor2, bonoValor2);
    CalcularValores1("AGILIDAD", numValor3, bonoValor3);
    CalcularValores1("VELOCIDAD", numValor4, bonoValor4);
    CalcularValores1("PODER", numValor5, bonoValor5);
    CalcularValores1("SENTIDOS", numValor6, bonoValor6);
    CalcularValores1("MEMORIA", numValor7, bonoValor7);
    CalcularValores1("PERSONALIDAD", numValor8, bonoValor8);
}
```

- **CalcularValores1():** Función usada para calcular un valor si aumentara en 1 punto y así poder mostrarlo en el menú.

```
void CalcularValores1(string Valor, Text valorText, Text bonoText)
{

    int Val = Personaje.Valor(Valor);
    int Bon = Personaje.BonoValor(Valor);
    int ValSum = Val + 1;
    int BonoSum;
    if (ValSum >= 8) { BonoSum = (ValSum - 8) / 2; }
    else { BonoSum = (ValSum - 9) / 2; }

    valorText.text = Val + " > " + ValSum;
    bonoText.text = Bon + " > " + BonoSum;
}
```

- **CalcularValores2():** Función usada para calcular un valor si aumentara en 2 puntos y así poder mostrarlo en el menú.

```
void CalcularValores2(string Valor, Text valorText, Text bonoText)
{

    int Val = Personaje.Valor(Valor);
    int Bon = Personaje.BonoValor(Valor);
    int ValSum = Val + 2;
    int BonoSum;
    if (ValSum >= 8) { BonoSum = (ValSum - 8) / 2; }
    else { BonoSum = (ValSum - 9) / 2; }

    valorText.text = Val + " > " + ValSum;
    bonoText.text = Bon + " > " + BonoSum;
}
```

6.4.4. Selección de clase

Corresponde a la finalización del paso 4 de la creación de un personaje. En esta "pantalla", el usuario debe seleccionar la clase a la que quiere que su personaje pertenezca. Ver **Figura 6.4.4/1**.

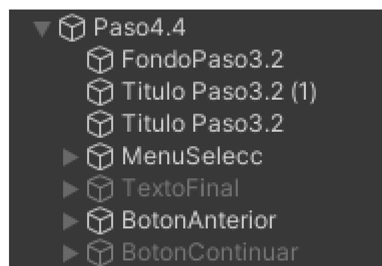

Figura 6.4.4/1: Captura de la jerarquía de la pantalla de selección de clase.

Esta "pantalla" está dividida en dos partes que se muestran dependiendo de si la selección está en proceso o ha terminado. En este paso, esas dos partes se llaman "MenuSelecc" y "TextoFinal". Como se puede observar, el botón para seguir a la siguiente "pantalla" está bloqueado, y no se desbloquea hasta que el proceso haya terminado por completo.

En la primera parte, se muestran las diferentes clases disponibles de Animaia en botones. Cuando el usuario pulse uno se esos botones, se mostrará el menú pop-up "MenuClase", donde el usuario podrá ver información de la clase seleccionada. Ver **Figura 6.4.4/2.**

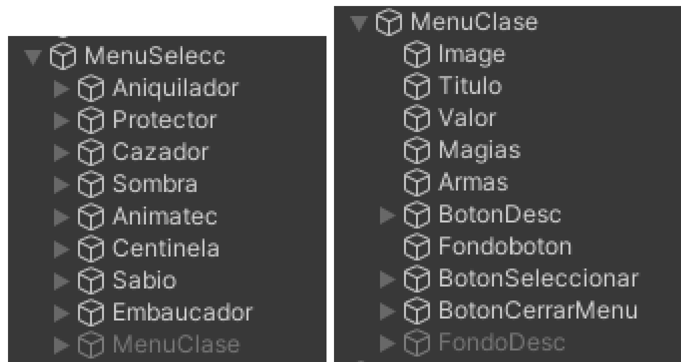

Figura 6.4.4/2: A la izquierda, captura de la jerarquía de la parte de las clases. A la derecha, captura de la jerarquía del menú de la clase seleccionada.

En este menú, el usuario puede confirmar la selección pulsando el botón "BotonSeleccionar". Cuando se selecciona una clase, se oculta la primera parte y se muestra la segunda parte. En esta, se muestra un simple mensaje recordando la selección hecha. También se desbloquea el botón para continuar a la siguiente pantalla. Ver **Figura 6.4.4/3**.

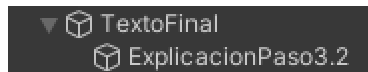

Figura 6.4.4/3: Captura de la jerarquía de la parte final.

La funcionalidad de esta pantalla proviene del script llamado "Paso 4_4". Ver **Figura 6.4.4/4**.

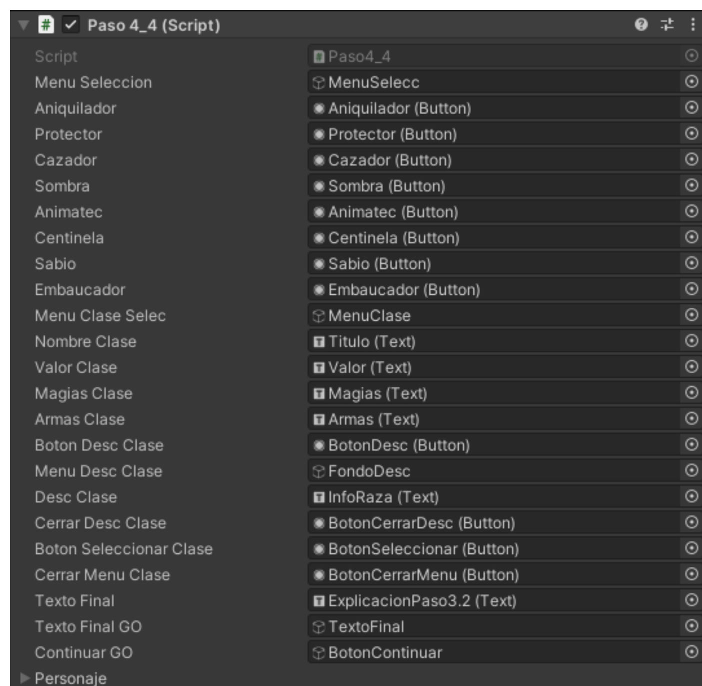

Figura 6.4.4/4: Captura del script Paso 4_4 y sus atributos.

Sus funciones son:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder manipular los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **Start():** Esta función se ejecuta en el primer frame en el que se activa la escena. Llama las funciones "IniciarBotonesClases" y "IniciarBotonesMenu" para inicializar y asignar a todos los botones de la "pantalla" sus funciones para cuando son pulsados.

```
void Start()
{
    IniciarBotonesClases();
    IniciarBotonesMenu();
}
```

- **IniciarBotonesClases():** Inicializa y asigna a todos los botones de cada clase sus funciones para cuando son pulsados.

```
void IniciarBotonesClases()
{
    Aniquilador.onClick.AddListener(AbrirAniq);
    Protector.onClick.AddListener(AbrirProt);
    Cazador.onClick.AddListener(AbrirCaza);
    Sombra.onClick.AddListener(AbrirSomb);
    Animatec.onClick.AddListener(AbrirAnim);
    Centinela.onClick.AddListener(AbrirCent);
    Sabio.onClick.AddListener(AbrirSabio);
    Embaucador.onClick.AddListener(AbrirEmba);
}
```

- **IniciarBotonesMenu():** Inicializa y asigna a todos los botones del menú de las clases sus funciones para cuando son pulsados.

```
void IniciarBotonesMenu()
{
    BotonDescClase.onClick.AddListener(AbrirDesc);
    CerrarDescClase.onClick.AddListener(CerrarDesc);
    BotonSeleccionarClase.onClick.AddListener(SelecClase);
    CerrarMenuClase.onClick.AddListener(CerrarClase);
}
```

- **Abrir*Clase*():** Tipo de función usada para preparar la información dentro del menú de la clase según la que se haya seleccionado. Cada clase tiene su propia función, pero todas comparten la misma estructura. Cuando termina de manipular los elementos del menú, lo activa para que sea visible. La parte en rojo del nombre de la función sustituye al nombre de cada clase.

```
void AbrirAniq()
{
    NombreClase.text = "Aniquilador";
    ValorClase.text = "Valor Vinculado: +2 a la Fuerza";
    MagiasClase.text = "Habilidades disponibles: Bala, Carga, Atq.
                        Expansivo, Obliteración, Enfurecer, Aterrar, Grito
                        de Guerra, Aligerar.";
    ArmasClase.text = "Armas disponibles: Cuerpo, Espada recta, Espadón,
                       Alabarda, Hacha, Gran Hacha.";
    DescClase.text = "Viven por y para el combate cuerpo a cuerpo. Su
                      extraordinaria fuerza y agresividad les permite
                      destrozar a sus oponentes en devastadores ataques que
                      diezman sus defensas. No hay protección posible ante
                      un aniquilador.";
    MenuClaseSelec.SetActive(true);
}
```

- **AbrirDesc():** Activa el objeto donde se encuentra la descripción de la clase para que esta se vea.

```
    void AbrirDesc()
    {
        MenuDescClase.SetActive(true);
    }
```

- **CerrarDesc():** Desactiva el objeto donde se encuentra la descripción de la clase para que esta se deje de ver.

```
    void CerrarDesc()
    {
        MenuDescClase.SetActive(false);
    }
```

- **CerrarClase():** Desactiva el menú de la clase para que deje de verse.

```
    void CerrarClase()
    {
        MenuClaseSelec.SetActive(false);
    }
```

- **SelecClase():** Función usada cuando se selecciona una clase para asignarla al personaje junto a todos los datos de la misma clase. Cuando termina de guardar esos datos, procede a terminar la parte de selección y muestra la parte final de la "pantalla", desbloqueando a la vez el botón para continuar al siguiente paso de la creación del personaje.

```
void SelecClase()
    {
        Personaje._clase.nombreClase = NombreClase.text;
        if(NombreClase.text == "Aniquilador")
        {
            AsignarClase("Aniquilador", "FUERZA");
            EmpezarHechizAniquilador();
            EmpezarArmasAniquilador();
        }
        else if(NombreClase.text == "Protector")
        {
            AsignarClase("Protector", "RESISTENCIA");
            EmpezarHechizProtector();
            EmpezarArmasProtector();
```

```
        }
        else if (NombreClase.text == "Cazador")
        {
            AsignarClase("Cazador", "AGILIDAD");
            EmpezarHechizCazador();
            EmpezarArmasCazador();
        }
        else if (NombreClase.text == "Sombra")
        {
            AsignarClase("Sombra", "VELOCIDAD");
            EmpezarHechizSombra();
            EmpezarArmasSombra();
        }
        else if (NombreClase.text == "Animatec")
        {
            AsignarClase("Animatec", "PODER");
            EmpezarHechizAnimatec();
            EmpezarArmasAnimatec();
        }
        else if (NombreClase.text == "Centinela")
        {
            AsignarClase("Centinela", "SENTIDOS");
            EmpezarHechizCentinela();
            EmpezarArmasCentinela();
        }
        else if (NombreClase.text == "Sabio")
        {
            AsignarClase("Sabio", "MEMORIA");
            EmpezarHechizSabio();
            EmpezarArmasSabio();
        }
        else if (NombreClase.text == "Embaucador")
        {
            AsignarClase("Embaucador", "PERSONALIDAD");
            EmpezarHechizEmbaucador();
            EmpezarArmasEmbaucador();
        }
        MenuClaseSelec.SetActive(false);
        MenuSeleccion.SetActive(false);
        TextoFinalGO.SetActive(true);
        ContinuarGO.SetActive(true);
    }
```

- **AsignarClase():** Función usada para asignar una clase al personaje, actualizar el valor vinculado a esa clase y editar el texto de la parte final de la “pantalla” para que muestre la clase seleccionada por el usuario.

```
void AsignarClase(string nomClase, string valorClase)
{
    Personaje._clase.valorClase = valorClase;
    Personaje.ActualizarValor(valorClase, 2);
    Personaje._clase.descClase = DescClase.text;
    TextoFinal.text = "Clase seleccionada: \n\n" + nomClase;
}
```

- **Crear*Arma*():** Grupo de funciones que comparten el mismo uso, generar un arma. Se usan cuando se genera el listado de armas disponibles de la clase que se selecciona. Existe una función por arma existente en este sistema de rol, y cada clase crea sus armas vinculadas. La parte en rojo del nombre de la función sustituye al nombre de cada arma.

```
Arma CrearCuerpo()
{
    return new Arma("Cuerpo", "El uso de puñetazos, codazos y patadas para
                    atacar.", 6, "FUERZA");
}
```

- **Crear*Habilidad*():** Grupo de funciones que comparten el mismo uso, generar una habilidad. Se usan cuando se genera el listado de habilidades disponibles de la clase que se selecciona. Existe una función por habilidad existente en este sistema de rol, y cada clase crea sus habilidades vinculadas. La parte en rojo del nombre de la función sustituye al nombre de cada habilidad.

```
Hechizo CrearBala()
{
    return new Hechizo("Bala", "Concentra una pequeña parte de su alma en un
                        punto y la dispara a alta velocidad para infligir
                        daño a un enemigo.", 6, 6, "Magia");
}
```

- **EmpezarArmas*Clase*():** Grupo de funciones que comparten el mismo uso, generar el listado de armas vinculadas de una clase. Existe una función por cada clase. La parte en rojo del nombre de la función sustituye al nombre de cada clase.

```
void EmpezarArmasAniquilador()
{
    Arma arma1 = CrearCuerpo();
    Arma arma2 = CrearEspRecta();
    Arma arma3 = CrearEspadon();
    Arma arma4 = CrearAlabarda();
    Arma arma5 = CrearHacha();
    Arma arma6 = CrearGHacha();
    Personaje._clase.listaArmasClase = new Arma[] { arma1, arma2, arma3,
                                                    arma4, arma5, arma6 };
}
```

- **EmpezarHechiz*Clase*():** Grupo de funciones que comparten el mismo uso, generar el listado de habilidades vinculadas de una clase. Existe una función por cada clase. La parte en rojo del nombre de la función sustituye al nombre de cada clase.

```
void EmpezarHechizAniquilador()
{
    Hechizo hech1 = CrearBala();
    Hechizo hech2 = CrearCarga();
    Hechizo hech3 = CrearExpansivo();
    Hechizo hech4 = CrearObliteracion();
    Hechizo hech5 = CrearEnfurecer();
    Hechizo hech6 = CrearAterrar();
    Hechizo hech7 = CrearGrito();
    Hechizo hech8 = CrearAligerar();
    Personaje._clase.listaHechizosClase = new Hechizo[] { hech1, hech2,
                                          hech3, hech4, hech5, hech6, hech7,
                                          hech8 };
}
```

6.4.5. Selección del arma

Corresponde a la finalización del paso 5 de la creación de un personaje. En esta “pantalla”, el usuario debe seleccionar el arma que quiere usar durante las partidas. Ver **Figura 6.4.5/1**.

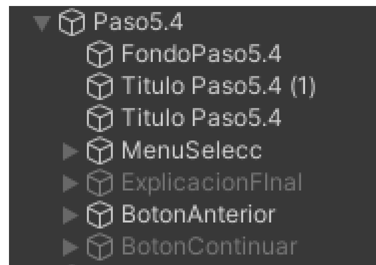

Figura 6.4.5/1: Captura de la jerarquía de la pantalla de selección de arma.

Esta "pantalla" está dividida en dos partes que se muestran dependiendo de si la selección está en proceso o ha terminado. En este paso, esas dos partes se llaman "MenuSelecc" y "ExplicacionFinal". Como se puede observar, el botón para seguir a la siguiente "pantalla" está bloqueado, y no se desbloquea hasta que el proceso haya terminado por completo.

En la primera parte, se muestran las diferentes armas disponibles que dependen de la clase que se haya seleccionado en el paso anterior. Cuando el usuario pulse uno se esos botones, se mostrará el menú pop-up "MenuArma", donde el usuario podrá ver información del arma seleccionada. Ver **Figura 6.4.5/2.**

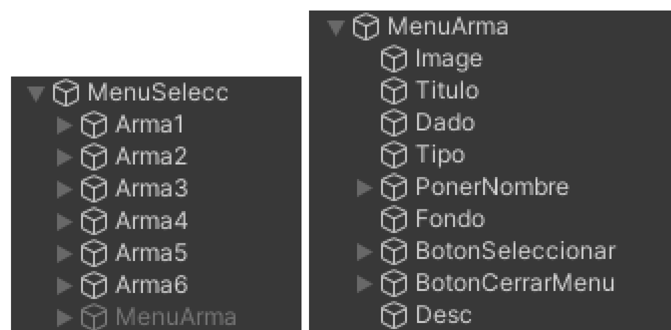

Figura 6.4.5/2: A la izquierda, captura de la jerarquía de la parte de las armas. A la derecha, captura de la jerarquía del menú del arma seleccionada.

En este menú, el usuario puede confirmar la selección pulsando el botón "BotonSeleccionar". Cuando se selecciona un arma, se oculta la primera parte y se muestra la segunda parte. En esta, se muestra un simple mensaje recordando la selección hecha. También se desbloquea el botón para continuar a la siguiente pantalla. Ver **Figura 6.4.5/3**.

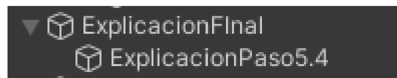

Figura 6.4.5/3: Captura de la jerarquía de la parte final.

La funcionalidad de esta pantalla proviene del script llamado “Paso 5_4”. Ver **Figura 6.4.5/4**.

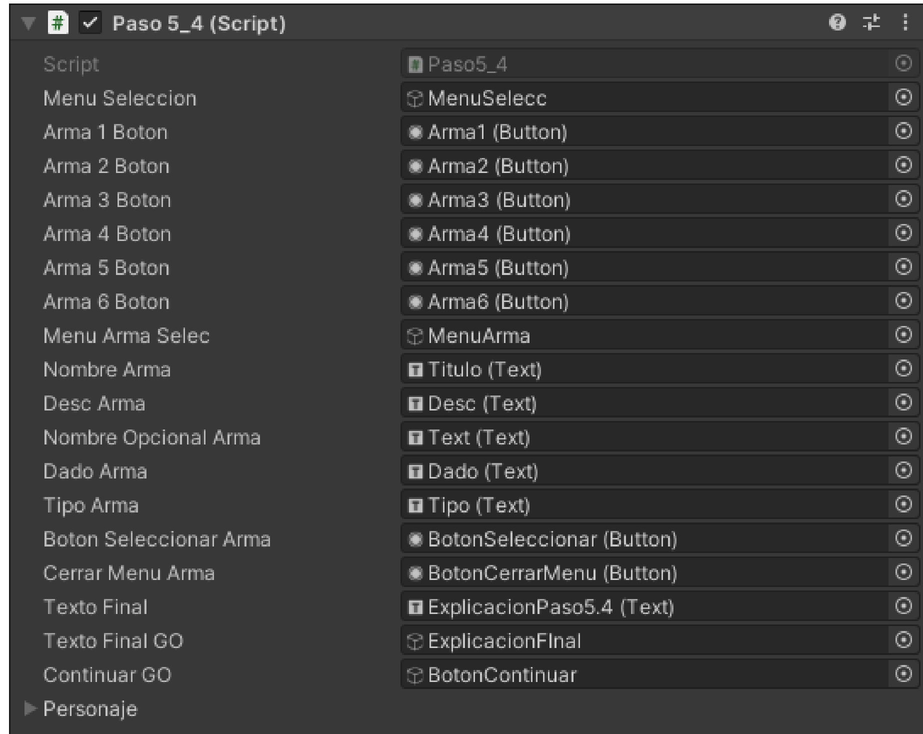

Figura 6.4.5/4: Captura del script Paso 5_4 y sus atributos.

Sus funciones son:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder manipular los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **Start():** Esta función se ejecuta en el primer frame en el que se activa la escena. Llama las funciones "IniciarBotonesArma" y "IniciarBotonesMenu" para inicializar y asignar a todos los botones de la "pantalla" sus funciones para cuando son pulsados.

```
void Start()
{
    IniciarBotonesArma();
    IniciarBotonesMenu();
}
```

- **IniciarBotonesArma():** Inicializa y asigna a todos los botones de cada arma sus funciones para cuando son pulsados.

```
void IniciarBotonesArma()
{
    Arma1Boton.onClick.AddListener(AbrirMenu1);
    Arma2Boton.onClick.AddListener(AbrirMenu2);
    Arma3Boton.onClick.AddListener(AbrirMenu3);
    Arma4Boton.onClick.AddListener(AbrirMenu4);
    Arma5Boton.onClick.AddListener(AbrirMenu5);
    Arma6Boton.onClick.AddListener(AbrirMenu6);
}
```

- **AbrirMenu() 1-6:** Función que se ejecuta cuando se pulsa el botón de un arma. Es usada para preparar el menú del arma seleccionada con toda la información referente a ese arma. Cuando termina de manipular los elementos del menú, entonces lo activa para que se haga visible.

```
void AbrirMenu1()
{
    dado = Personaje._clase.listaArmasClase[0].dadoArma;
    tipo = Personaje._clase.listaArmasClase[0].tipoArma;
    NombreArma.text = Personaje._clase.listaArmasClase[0].nombreArma;
    DescArma.text = Personaje._clase.listaArmasClase[0].descArma;
    DadoArma.text = "Dado Vinculante:\n d" + dado;
    TipoArma.text = "Tipo Daño:\n " + tipo;
    MenuArmaSelec.SetActive(true);
}
```

- **IniciarBotonesMenu():** Inicializa y asigna a todos los botones del menú de las clases sus funciones para cuando son pulsados.

```
void IniciarBotonesMenu()
{
    BotonSeleccionarArma.onClick.AddListener(SelecArma);
    CerrarMenuArma.onClick.AddListener(CerrarArma);
}
```

- **CerrarArma():** Desactiva el menú del arma para que deje de verse.

```
void IniciarBotonesMenu()
{
    BotonSeleccionarArma.onClick.AddListener(SelecArma);
    CerrarMenuArma.onClick.AddListener(CerrarArma);
}
```

- **SelecArma():** Función usada cuando se confirma la selección de un arma. La información de esta se vincula al arma equipada del personaje. También le asigna un nombre si el usuario decide ponerle un nombre personalizado al arma. Finalmente, calcula el daño que puede hacer el ataque de esa arma para poder mostrarlo en el texto final, termina con la primera parte de la “pantalla” desactivándola y activa la segunda parte para que se vea la elección y el cálculo de daño del arma seleccionada. También activa el botón para continuar al siguiente paso.

```
void SelecArma()
{
    Personaje.ArmaEquipada.descArma = DescArma.text;
    Personaje.ArmaEquipada.dadoArma = dado;
    Personaje.ArmaEquipada.tipoArma = tipo;
    Debug.Log(NombreOpcionalArma.text);
    if(NombreOpcionalArma.text == "")
    {
        Personaje.ArmaEquipada.nombreArma = NombreArma.text;
    }
    else
    {
        Personaje.ArmaEquipada.nombreArma = NombreOpcionalArma.text;
    }
    MenuArmaSelec.SetActive(false);
    MenuSeleccion.SetActive(false);
    int bono;
    string calculo = " ";
    if (tipo == "FUERZA")
    {
        bono = Personaje.BonoValor("FUERZA");
        if (bono >= 0) { calculo = "d" + dado + "+" + bono; }
        else { calculo = "d" + dado + bono; }
    }
    else if (tipo == "AGILIDAD")
    {
        bono = Personaje.BonoValor("AGILIDAD");
        if (bono >= 0) { calculo = "d" + dado + "+" + bono; }
        else { calculo = "d" + dado + bono; }
    }
    TextoFinal.text = "Arma seleccionada:\n" +
                      Personaje.ArmaEquipada.nombreArma + "\n\nCálculo
                      actual de daño:\n" + calculo;
    TextoFinalGO.SetActive(true);
    ContinuarGO.SetActive(true);
    //DataPersistanceManager.instance.ActualizarPersonaje(Personaje);
}
```

6.4.6. Asignación de acciones ventaja

Corresponde a la finalización del paso 6 de la creación de un personaje. En esta “pantalla”, el usuario debe seleccionar 4 acciones con las que quiere que su personaje sea aventajado. Ver **Figura 6.4.6/1**.

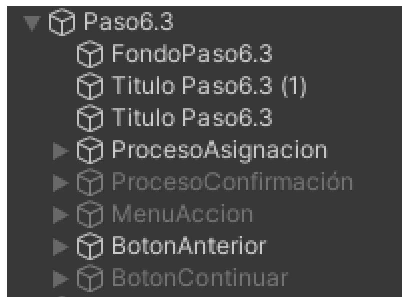

Figura 6.4.6/1: Captura de la jerarquía de la pantalla de selección de acciones ventaja.

Esta "pantalla" está dividida en dos partes que se muestran dependiendo de si la selección está en proceso o ha terminado. En este paso, esas dos partes se llaman "ProcesoAsignacion" y "ProcesoConfirmación". Como se puede observar, el botón para seguir a la siguiente "pantalla" está bloqueado, y no se desbloquea hasta que el proceso haya terminado por completo.

En la primera parte, se muestran todas las acciones disponibles del sistema de rol. Cuando el usuario pulse una de esas acciones, se mostrará el menú pop-up "MenuAccion", donde el usuario podrá ver información de la acción seleccionada. Ver **Figura 6.4.6/2.**

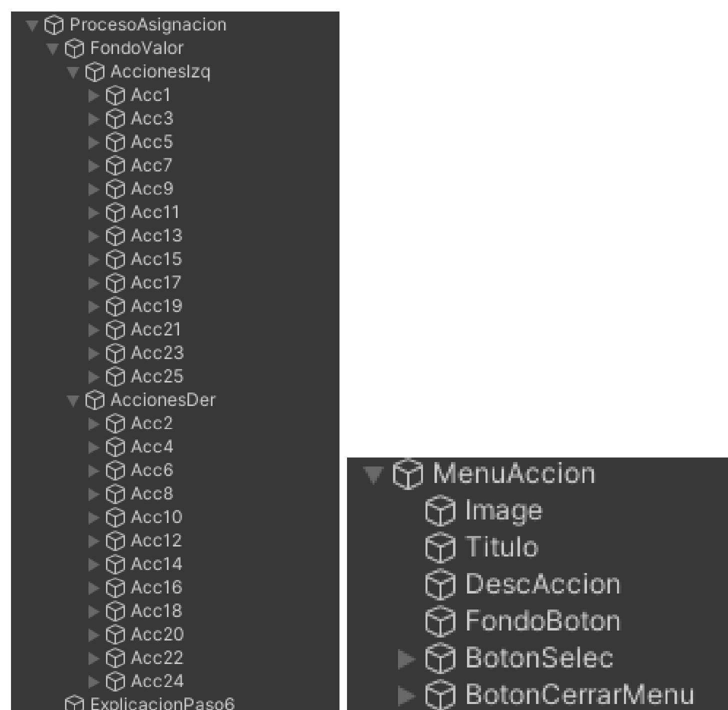

Figura 6.4.6/2: A la izquierda, captura de la jerarquía de la parte de las acciones. A la derecha, captura de la jerarquía del menú de la acción seleccionada.

En este menú, el usuario puede confirmar la selección pulsando el botón “BotonSelec”. Cuando el usuario haya terminado de seleccionar todas las acciones ventaja, entonces se oculta la primera parte y se muestra la segunda parte. En esta, se muestra cada acción seleccionada y se muestra una pregunta de confirmación de la selección para el usuario. Ver **Figura 6.4.6/3**.

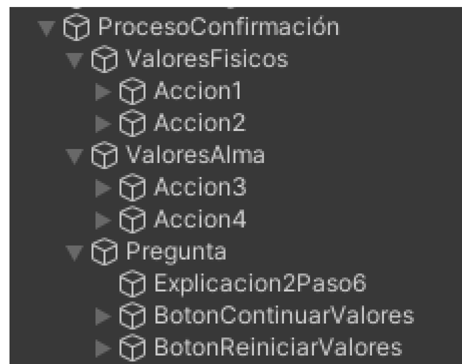

Figura 6.4.6/3: Captura de la jerarquía de la parte final.

La funcionalidad de esta pantalla proviene del script llamado “Paso 6_3”. Ver **Figura 6.4.6/4**.

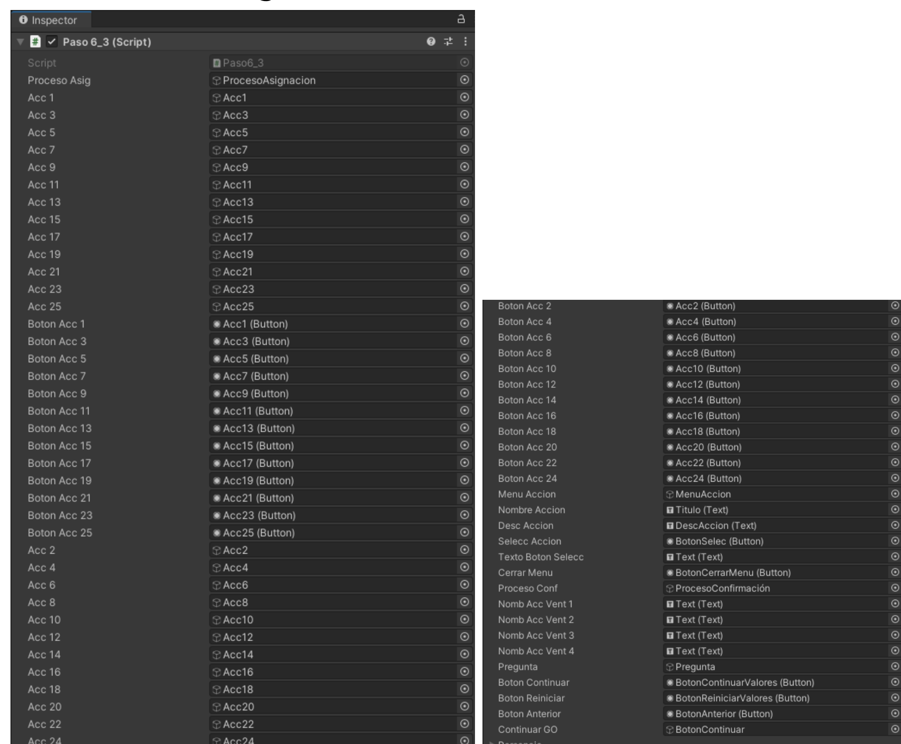

Figura 6.4.6/4: Capturas del script Paso 6_3 y sus atributos.

Sus funciones son:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder manipular los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **Start():** Inicializa y asigna a todos los botones de la "pantalla" sus funciones para cuando son pulsados.

```
void Start()
{
    numAcc = 0;
    IniciarBotonesAcciones();
    SeleccAccion.onClick.AddListener(SeleccAccionVent);
    CerrarMenu.onClick.AddListener(CerrarMenuAccion);
    BotonContinuar.onClick.AddListener(AsignarAcciones);
    BotonReiniciar.onClick.AddListener(ReiniciarProceso);
    botonAnterior.onClick.AddListener(ReiniciarProceso);
}
```

- **IniciarBotonesAcciones():** Inicializa y asigna a los botones correspondientes a las acciones sus funciones para cuando son pulsados.

```
void IniciarBotonesAcciones()
{
    BotonAcc1.onClick.AddListener(Boton1);
    BotonAcc2.onClick.AddListener(Boton2);
    BotonAcc3.onClick.AddListener(Boton3);
    BotonAcc4.onClick.AddListener(Boton4);
    BotonAcc5.onClick.AddListener(Boton5);
    BotonAcc6.onClick.AddListener(Boton6);
    BotonAcc7.onClick.AddListener(Boton7);
    BotonAcc8.onClick.AddListener(Boton8);
    BotonAcc9.onClick.AddListener(Boton9);
    BotonAcc10.onClick.AddListener(Boton10);
    BotonAcc11.onClick.AddListener(Boton11);
    BotonAcc12.onClick.AddListener(Boton12);
    BotonAcc13.onClick.AddListener(Boton13);
    BotonAcc14.onClick.AddListener(Boton14);
    BotonAcc15.onClick.AddListener(Boton15);
    BotonAcc16.onClick.AddListener(Boton16);
    BotonAcc17.onClick.AddListener(Boton17);
    BotonAcc18.onClick.AddListener(Boton18);
    BotonAcc19.onClick.AddListener(Boton19);
    BotonAcc20.onClick.AddListener(Boton20);
    BotonAcc21.onClick.AddListener(Boton21);
    BotonAcc22.onClick.AddListener(Boton22);
    BotonAcc23.onClick.AddListener(Boton23);
    BotonAcc24.onClick.AddListener(Boton24);
    BotonAcc25.onClick.AddListener(Boton25);
}
```

- **Boton() 1-25:** Conjunto de funciones con el mismo cometido. Se activan al pulsar el botón correspondiente a una acción. Llaman a la función “AbrirMenu” pasando como parámetro la información de la acción a la que pertenecen.

```
void Boton1()
{
    AbrirMenu   ("PERCIBIR",
                "Vinculada al atributo SENTIDOS. Simboliza la acción de
                 mirar a algo, vigilar, inspeccionar con la mirada o buscar
                 algo por el entorno. Todo lo que sea usar la visión para
                 algo en específico entra dentro de esta acción.", acc1);
}
```

- **void AbrirMenu():** Función usada para preparar el menú de la acción con su respectiva información. Cuando ya ha

manipulado todos los elementos necesarios, activa el menú para que sea visible.

```
void AbrirMenu(string nombre, string desc, GameObject boto)
{
    NombreAccion.text = nombre;
    TextoBotonSelecc.text = "Seleccionar " + nombre + " como acción
                             ventaja";
    DescAccion.text = desc;
    botonAct = boto;
    MenuAccion.SetActive(true);
}
```

- **void SeleccAccionVent():** Función usada para controlar el proceso de selección de las acciones ventaja. Se activa cada vez que el usuario selecciona una acción. Si se han seleccionado todas las acciones ventaja posibles, se encarga de terminar la primera parte del proceso y mostrar la segunda parte.

```
void SeleccAccionVent()
{
    if (numAcc == 0)
    {
        NombAccVent1.text = NombreAccion.text;
    }
    if (numAcc == 1)
    {
        NombAccVent2.text = NombreAccion.text;
    }
    if (numAcc == 2)
    {
        NombAccVent3.text = NombreAccion.text;
    }
    if (numAcc == 3)
    {
        NombAccVent4.text = NombreAccion.text;
        ProcesoAsig.SetActive(false);
        ProcesoConf.SetActive(true);
    }
    botonAct.SetActive(false);
    numAcc++;
    MenuAccion.SetActive(false);
}
```

- **void AsignarAcciones():** Función usada para asignar las acciones ventaja seleccionadas al personaje, terminando el proceso. Se activa cuando el usuario confirma la selección en la segunda parte del proceso, activando el botón que permite seguir hasta el final de la creación del personaje.

```
void AsignarAcciones()
{
    Personaje.accVent1 = NombAccVent1.text;
    Personaje.accVent2 = NombAccVent2.text;
    Personaje.accVent3 = NombAccVent3.text;
    Personaje.accVent4 = NombAccVent4.text;
    ContinuarGO.SetActive(true);
    pregunta.SetActive(false);
}
```

- **ReiniciarProceso():** Devuelve al estado predeterminado todos los elementos de la “pantalla” cuando el usuario decide reiniciar el proceso en la segunda parte. Esto por defecto, devuelve el proceso de selección de acciones ventaja al principio.

```
void ReiniciarProceso()
{
    numAcc = 0;
    acc1.SetActive(true);
    acc3.SetActive(true);
    acc5.SetActive(true);
    acc7.SetActive(true);
    acc9.SetActive(true);
    acc11.SetActive(true);
    acc13.SetActive(true);
    acc15.SetActive(true);
    acc17.SetActive(true);
    acc19.SetActive(true);
    acc21.SetActive(true);
    acc23.SetActive(true);
    acc25.SetActive(true);
    acc2.SetActive(true);
    acc4.SetActive(true);
    acc6.SetActive(true);
    acc8.SetActive(true);
    acc10.SetActive(true);
    acc12.SetActive(true);
    acc14.SetActive(true);
```

```
        acc16.SetActive(true);
        acc18.SetActive(true);
        acc20.SetActive(true);
        acc22.SetActive(true);
        acc24.SetActive(true);
        pregunta.SetActive(true);
        MenuAccion.SetActive(false);
        ProcesoAsig.SetActive(true);
        ProcesoConf.SetActive(false);
        ContinuarGO.SetActive(false);
    }
```

- **CerrarMenuAccion():** Desactiva el menú de una acción para que deje de verse.

```
    void CerrarMenuAccion()
    {
        MenuAccion.SetActive(false);
    }
```

6.4.7. Paso final

Corresponde a la finalización del paso 7 y final de la creación de un personaje. En esta "pantalla", el usuario debe introducir el nombre del personaje. Ver **Figura 6.4.7/1**.

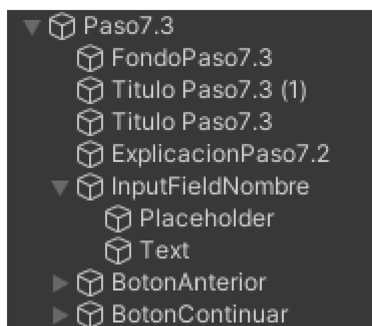

Figura 6.4.7/1: Captura de la jerarquía de la pantalla final del creador de personajes.

Esta "pantalla" sólo contiene texto explicando el final del creador y un cuadro de texto llamado "InputFieldNombre" donde el usuario puede introducir el nombre del personaje.

La funcionalidad de esta pantalla proviene del script llamado "Paso 7". Ver **Figura 6.4.7/2**.

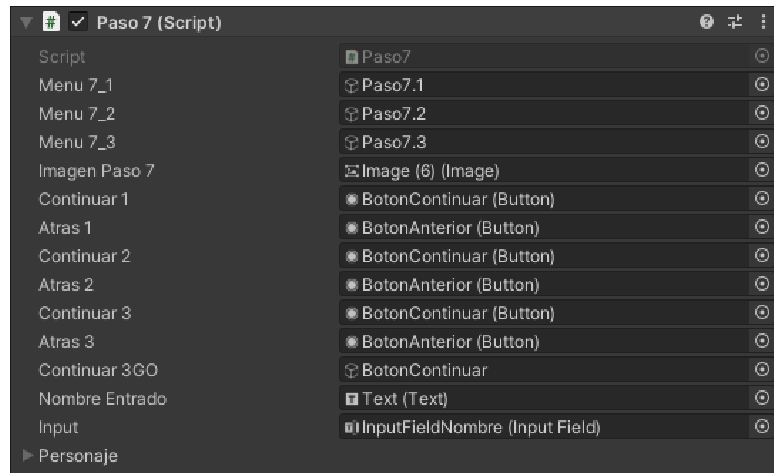

Figura 6.4.7/2: Capturas del script Paso 7 y sus atributos.

Sus funciones son del estilo vistas en el apartado 6.4.1, las que permiten avanzar y retroceder entre "pantallas" del creador de personajes, con la excepción de unas pocas funciones. Estas son:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder manipular los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **Update():** Función que se ejecuta a cada frame. Sirve para controlar que el usuario ha entrado un nombre para el personaje en el objeto "InputFieldNombre". Si es así, desbloquea el botón para finalizar la creación y que lleva de vuelta el usuario al menú inicial.

```
void Update()
{
    if(Input.text != "" && Input.text != " ")
    {
        continuar3GO.SetActive(true);
    }
    else
    {
        continuar3GO.SetActive(false);
    }
}
```

6.5. Implementación del gestor de personaje

El gestor de personaje es donde el usuario puede consultar y editar todos los atributos de sus personajes. Para cumplir con ese cometido, se ha implementado una escena que contiene toda la información del personaje, organizada en diferentes bloques, y donde el usuario es capaz de modificar múltiples elementos del personaje. También se han implementado mecánicas que automatizan varios aspectos de las partidas de rol.

Para implementar un gestor que simule y automatice mecánicas respecto a las clásicas fichas de personajes, cada aspecto principal del personaje se ha organizado en bloques diferentes, que se mostrarán mediante el uso de un botón desplegable. En cada bloque están situados los diferentes tipos de información que conforman los personajes y las mecánicas que rodean cada tipo de dato.

Todos los scripts que controlan el comportamiento de todos los elementos del gestor se encuentran en dos objetos de la jerarquía llamados “Funcionabilidad” y “DatosPersonaje”. En este último solo se encuentran los scripts que controlan los elementos del bloque “Equipo”.

6.5.1. Navegación y organización

En esta escena todos los elementos que lo conforman están en un *canvas*. Ese canvas está implementado de la misma forma que el del menú inicial, explicado en el punto 6.3.

El gestor de personajes está dividido en 2 partes, la superior e inferior. En la superior está la parte donde los usuarios pueden

acceder a la información básica del personaje, como la clase, la raza o el nivel y la experiencia acumulada. En la parte inferior es donde la información está organizada en diferentes bloques. Estos bloques son:

- **Valores:** Bloque donde los valores del personaje se muestran y se pueden modificar.
- **Acciones:** Bloque donde el usuario puede realizar las tiradas de todo tipo de acciones que el personaje pueda tomar.
- **Habilidades:** Bloque usado exclusivamente para la gestión de las habilidades del personaje.
- **Equipo:** Bloque que representa el inventario del personaje, todo con lo que esté equipado, está estipulado aquí.
- **Características:** Bloque de información referente a la personalidad y aspectos enfocados a la interpretación del personaje.
- **Manual del sistema:** Bloque donde se puede encontrar el manual del sistema entero por si se necesita consultar información durante las partidas.

La explicación en profundidad de la implementación de cada bloque se podrá encontrar en los próximos apartados.

Cada bloque ocupa el mismo espacio en la pantalla, y solo se puede mostrar un solo bloque a la vez. La visibilidad de cada bloque se elige mediante el uso de un botón desplegable que se encuentra entre la parte superior e inferior del gestor. Ver **Figura 6.5.1/1**.

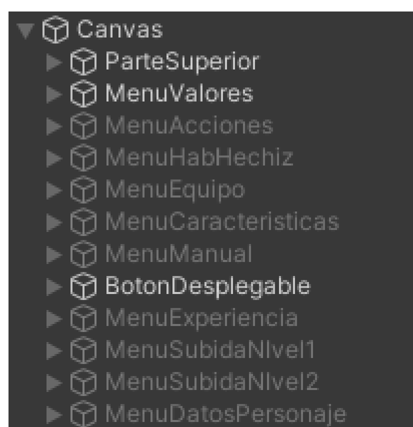

Figura 6.5.1/1: Captura de la jerarquía del *canvas* de la escena.

Este botón desplegable muestra una opción por cada bloque existente, y cuando se selecciona una opción, se muestra el bloque seleccionado, el título del botón desplegable cambia al nombre del bloque actual y las opciones vuelven a desaparecer. Ver **Figura 6.5.1/2**.

Figura 6.5.1/2: Captura de la jerarquía del botón desplegable cuando está activo.

El funcionamiento del botón desplegable está controlado mediante el script "Desplegable". Ver **Figura 6.5.1/3**.

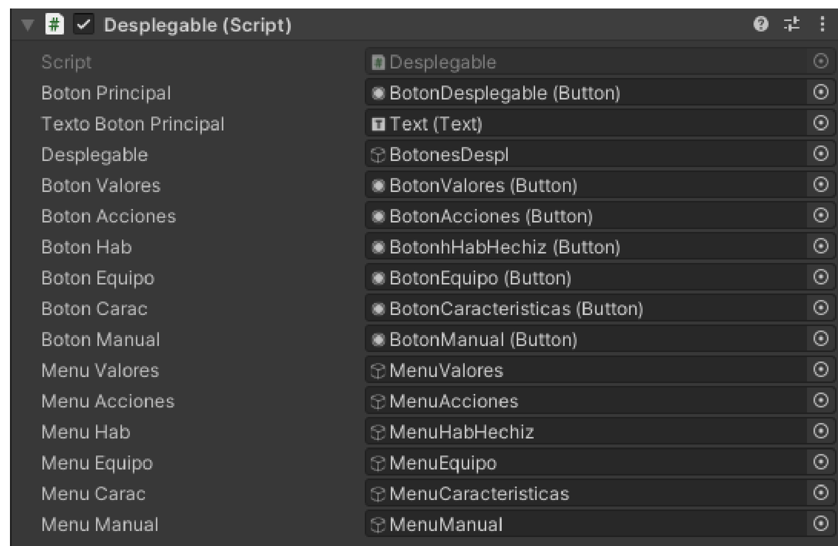

Figura 6.5.1/3: Captura del script en el inspector de Unity que controla el botón desplegable.

Sus funciones son las siguientes:

- **Start():** Función ejecutada en el primer frame al activarse la escena. Inicializa y asigna a todos los botones del desplegable sus funciones para cuando son pulsados. También manipula la visibilidad de los elementos, tanto

propios del desplegable, como los bloques de información de la parte inferior del gestor para dejarlos en su forma predeterminada.

```
void Start()
{
    botonPrincipal.onClick.AddListener(PrepararDesp);
    botonValores.onClick.AddListener(Bypass1);
    botonAcciones.onClick.AddListener(Bypass2);
    botonHab.onClick.AddListener(Bypass3);
    botonEquipo.onClick.AddListener(Bypass4);
    botonCarac.onClick.AddListener(Bypass5);
    botonManual.onClick.AddListener(Bypass6);

    desplegable.SetActive(false);
    menuValores.SetActive(false);
    menuAcciones.SetActive(false);
    menuHab.SetActive(false);
    menuEquipo.SetActive(false);
    menuCarac.SetActive(false);
    menuManual.SetActive(false);
    menuValores.SetActive(true);
    textoBotonPrincipal.text = "VALORES";

    desplegableActivo = false;
}
```

- **Bypass() 1-6:** Grupo de funciones usado cuando se selecciona una opción del desplegable. Se ocupan de llamar la función "MostrarMenu" pasando por parámetro el bloque y el nombre seleccionados.

```
void Bypass1()
{
    MostrarMenu(menuValores,"VALORES");
}
```

- **MostrarMenu():** Función dedicada a activar el bloque seleccionado y a cambiar el nombre mostrado en el desplegable.

```
void MostrarMenu(GameObject menu, string nombreMenu)
{
    desplegable.SetActive(false);
    desplegableActivo = false;
    menuValores.SetActive(false);
    menuAcciones.SetActive(false);
    menuHab.SetActive(false);
    menuEquipo.SetActive(false);
    menuCarac.SetActive(false);
    menuManual.SetActive(false);
    menu.SetActive(true);
    textoBotonPrincipal.text = nombreMenu;
}
```

- **PrepararDesp():** Función usada cuando se pulsa el botón principal del desplegable. Controla la lógica del botón por si se tiene que desplegar o no.

```
void PrepararDesp()
{
    if (desplegableActivo)
    {
      desplegable.SetActive(false);
      desplegableActivo = false;
    }
    else { desplegable.SetActive(true); desplegableActivo = true; }
}
```

Los jugadores también son capaces de volver al menú inicial pulsando el botón que se encuentra a la parte superior derecha de la pantalla. El script que controla su comportamiento es el de la **Figura 6.5.1/4**.

Figura 6.5.1/4: Captura del script en el inspector de Unity que controla el botón para volver al menú inicial.

Este script solo contiene 2 funciones:

- **Start():** Función ejecutada en el primer frame al activarse la escena. Inicializa el botón y le asigna la función “Volver”.

```
void Start()
{
    botonVolver.onClick.AddListener(Volver);
}
```

- **Volver():** Función al pulsar el botón. Guarda los datos actuales del personaje activo y carga la escena del menú inicial.

```
void Volver()
{
    DataPersistanceManager.instance.GuardarPersonaje();
    SceneManager.LoadScene("Main Screen");
}
```

6.5.2. Sistema de progresión

Para permitir a los personajes subir de nivel, se ha creado un grupo de pasos y funciones que se encargan de ello. El usuario solo debe acceder al menú de experiencia e introducir los puntos de experiencia que obtiene el personaje. En caso de subir de nivel, el sistema se encarga de empezar el proceso de aumento de valores. Ese menú se puede acceder pulsando el botón donde se muestra el nivel y la experiencia del personaje, localizado en la parte superior derecha del gestor. Ver **Figura 6.5.2/1**.

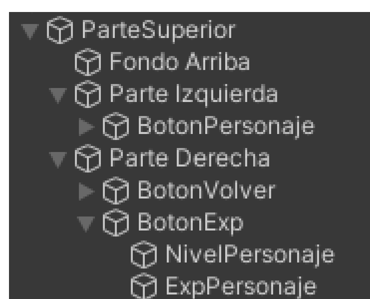

Figura 6.5.2/1: Captura de la jerarquía de la parte superior del gestor de personajes.

En ese menú se muestra una descripción de la experiencia, los puntos actuales, el límite del nivel y un cuadro de texto donde el usuario puede introducir los puntos de experiencia obtenidos. Ver **Figura 6.5.2/2**.

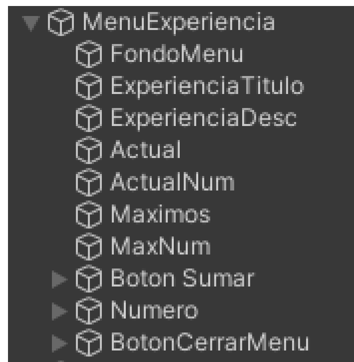

Figura 6.5.2/2: Captura de la jerarquía del menú de experiencia.

La funcionalidad tanto del botón como la del menú es controlada por el script “Boton Exp”. Ver **Figura 6.5.2/3**.

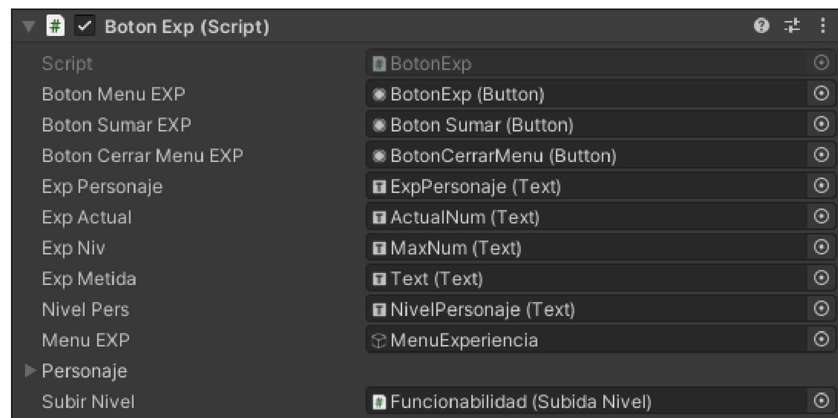

Figura 6.5.2/3: Captura del script en el inspector de Unity y sus atributos.

Sus funciones son:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder manipular los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **Start():** Inicializa y asigna a los botones, tanto el principal como los del menú sus funciones para cuando son pulsados. También inicializa los textos del menú y el botón para mostrar la información del personaje actual nada más cargar la escena.

```
void Start()
{
    botonMenuEXP.onClick.AddListener(AbrirMenuEXP);
    botonSumarEXP.onClick.AddListener(SumarEXP);
    botonCerrarMenuEXP.onClick.AddListener(CerrarMenuEXP);
    expActPersonaje = Personaje.exp;
    expNivPersonaje = Personaje.expProxNivel;
    expPersonaje.text = "EXP: " + expActPersonaje + "/" + expNivPersonaje;
    expActual.text = expActPersonaje.ToString();
    expNiv.text = expNivPersonaje.ToString();
}
```

- **AbrirMenuEXP():** Función usada cuando se pulsa el botón principal para abrir el menú de experiencia, activándolo.

```
void AbrirMenuEXP()
{
    menuEXP.SetActive(true);
}
```

- **CerrarMenuEXP():** Función usada para cerrar el menú de experiencia, desactivándolo.

```
void CerrarMenuEXP()
{
    menuEXP.SetActive(false);
}
```

- **SumarEXP():** Función usada para agregar nuevos puntos de experiencia al personaje, actualizando los textos que muestran esos puntos. Si esa experiencia rebasa el límite de nivel, empieza el proceso de subida de nivel llamando las funciones “SubirNivel” de la clase “Personaje” y del script “SubidaNivel”.

```
void SumarEXP()
{
    int expEntrada = int.Parse(expMetida.text);
    expActPersonaje = expActPersonaje + expEntrada;
    if (expActPersonaje >= expNivPersonaje)
    {
        expNivPersonaje = (expNivPersonaje * 2) + (expNivPersonaje / 2);
        Personaje.SubirNivel();
        subirNivel.SubirNivel();
        NivelPers.text = "Nivel: " + Personaje.nivel;
    }
    expActual.text = expActPersonaje.ToString();
    expNiv.text = expNivPersonaje.ToString();
    expPersonaje.text = "EXP: " + expActPersonaje + "/" + expNivPersonaje;
    Personaje.exp = expActPersonaje;
    Personaje.expProxNivel = expNivPersonaje;
}
```

Cuando un personaje alcanza los puntos necesarios para subir de nivel, el personaje aumenta en 2 puntos el valor vinculado a su clase y entonces el jugador debe elegir un valor vinculado a la raza a la que pertenece su personaje para aumentarla en un punto, y otro valor cualquiera para también aumentarlo en un punto. Para mostrar ese proceso, se ha separado en 2 pasos visibles, donde primero se muestra una ventana con los valores de la raza del personaje y que el usuario debe elegir. Ver **Figuras 6.5.2/4.**

Figura 6.5.2/4: Captura de la jerarquía de la primera ventana mostrada al subir de nivel.

Cuando el jugador elige uno de los valores de la raza, automáticamente se mostrará en el siguiente paso, una ventana con un menú, ya visto en la Figura **6.4.3/3**.

Como se ha visto en la última función mostrada, se hace uso de otro script, el “SubidaNivel”, que se encarga de controlar las ventanas de selección de los valores que el usuario puede aumentar y de realizar dichos aumentos de valores seleccionados. Ver Figuras **6.5.2/5A** y **6.5.2/5B**.

Figura 6.5.2/5A: Captura del script en el inspector de Unity y sus atributos.

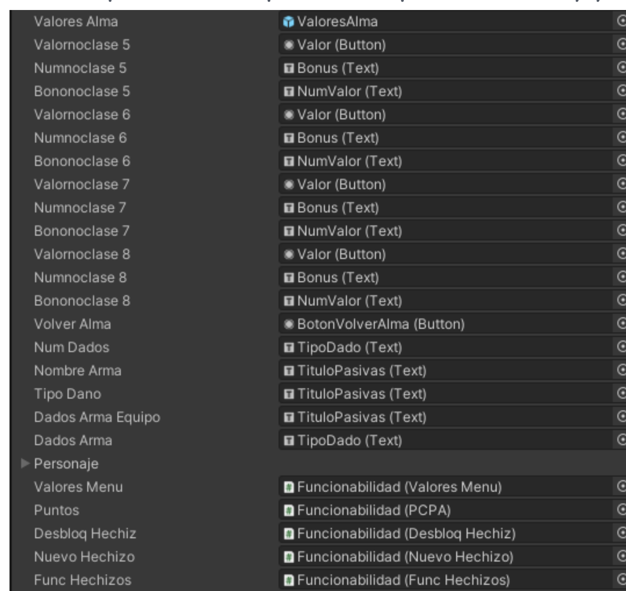

Figura 6.5.2/5B: Captura del script en el inspector de Unity y sus atributos.

Sus funciones son:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder manipular los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **Start():** Inicializa y asigna a todos los botones de las ventanas que se muestran sus funciones para cuando son pulsados durante el proceso. También inicializa los textos de las ventanas y de todos los valores que han cambiado por la función "SubirNivel" de la clase "Personaje" ejecutada anteriormente.

```
void Start()
{
    IniciarBotonesMenuClase();
    IniciarBotonesMenuResto();
    ValoresMenu.IniciarValores();
    Puntos.ActPuntos();
    numDados.text = Personaje._otros.numDados.ToString();
    nombreArma.text = Personaje.ArmaEquipada.nombreArma;
    tipoDano.text = Personaje.ArmaEquipada.tipoArma;
    dadosArmaEquipo.text = "d" + Personaje.ArmaEquipada.dadoArma;
    dadosArma.text = "d" + Personaje.ArmaEquipada.dadoArma;
    DesbloqHechiz.DesblHechiz();
    NuevoHechizo.IniciarMenuSelecc();
}
```

- **IniciarBotonesMenuClase():** A pesar del nombre, inicializa y asigna sus funciones a todos los botones de las ventana de selección de los valores de raza. Su nombre es debido a que, cuando se creó, inicializaba los botones correspondientes a los valores de la clase, pero se descartó la idea y el nombre permaneció.

```
void IniciarBotonesMenuClase()
//Cambiar nombre a IniciarBotonesMenuRaza despues de implementar todo
{
    valorRaza1.onClick.AddListener(Bypass1);
    valorRaza2.onClick.AddListener(Bypass2);
    valorRaza3.onClick.AddListener(Bypass3);
}
```

- **IniciarBotonesMenuResto():** Inicializa y asigna sus respectivas funciones al resto de botones del proceso de subida de nivel.

```
void IniciarBotonesMenuResto()
{
    valornoclase1.onClick.AddListener(Bypass4);
    valornoclase2.onClick.AddListener(Bypass5);
    valornoclase3.onClick.AddListener(Bypass6);
    valornoclase4.onClick.AddListener(Bypass7);
    valornoclase5.onClick.AddListener(Bypass8);
    valornoclase6.onClick.AddListener(Bypass9);
    valornoclase7.onClick.AddListener(Bypass10);
    valornoclase8.onClick.AddListener(Bypass11);
    valorCorporal.onClick.AddListener(AbrirValoresCuerpo);
    valorAlma.onClick.AddListener(AbrirValoresAlma);
    volverCorporal.onClick.AddListener(CerrarValoresCuerpo);
    volverAlma.onClick.AddListener(CerrarValoresAlma);
}
```

- **Bypass() 1-3**: Grupo de funciones usadas cuando se elige un valor de la raza del personaje. Se ocupan de actualizar el respectivo valor tanto en el personaje como los textos que lo muestran y muestran el segundo paso del proceso.

```
void Bypass1()
{
    Personaje.ActualizarValor(nombreRaza1.text, 1);
    ActMenuValores(nombreRaza1.text);
    SubirNivel2();
}
```

- **Bypass() 4-11**: Grupo de funciones usadas cuando se elige un valor en el paso final del proceso de subida de nivel. Se ocupan de actualizar el respectivo valor tanto

en el personaje como los textos que lo muestran y terminan el proceso.

```
void Bypass4()
{
    Personaje.ActualizarValor("FUERZA", 1);
    ActMenuValores("FUERZA");
    SubirNivel3();
}
```

- **SubirNivel()**: Función usada para iniciar el proceso de subida de nivel. Se encarga de actualizar el valor de la clase del personaje y de preparar la primera ventana con los valores base actuales vinculados a la raza. Cuando ya ha preparado los tres valores que muestra la ventana, la activa.

```
public void SubirNivel()
{
    Personaje.ActualizarValor(Personaje._clase.valorClase, 2);
    ActMenuValores(Personaje._clase.valorClase);

    //Cambiar nombres, numeros y bonos de cada valor
    nombreRaza1.text = Personaje.ElemRaza("valor1");
    int valorAct1 = Personaje.Valor(Personaje.ElemRaza("valor1"));
    int bonoAct1 = Personaje.BonoValor(Personaje.ElemRaza("valor1"));
    int valorSumado1 = valorAct1 + 1;
    int bonoSumado1;
    if (valorSumado1 >= 8) { bonoSumado1 = (valorSumado1 - 8) / 2; }
    else { bonoSumado1 = (valorSumado1 - 9) / 2; }
    numRaza1.text = Personaje.Valor(Personaje.ElemRaza("valor1")) + " > " +
                    valorSumado1;
    bonoRaza1.text = bonoAct1 + " > " + bonoSumado1;

    nombreRaza2.text = Personaje.ElemRaza("valor2");
    int valorAct2 = Personaje.Valor(Personaje.ElemRaza("valor2"));
    int bonoAct2 = Personaje.BonoValor(Personaje.ElemRaza("valor2"));
    int valorSumado2 = valorAct2 + 1;
    int bonoSumado2;
    if (valorSumado2 >= 8) { bonoSumado2 = (valorSumado2 - 8) / 2; }
    else { bonoSumado2 = (valorSumado2 - 9) / 2; }
    numRaza2.text = Personaje.Valor(Personaje.ElemRaza("valor2")) + " > " +
                    valorSumado2;
    bonoRaza2.text = bonoAct2 + " > " + bonoSumado2;

    nombreRaza3.text = Personaje.ElemRaza("valor3");
    int valorAct3 = Personaje.Valor(Personaje.ElemRaza("valor3"));
```

```
        int bonoAct3 = Personaje.BonoValor(Personaje.ElemRaza("valor3"));
        int valorSumado3 = valorAct3 + 1;
        int bonoSumado3;
        if (valorSumado3 >= 8) { bonoSumado3 = (valorSumado3 - 8) / 2; }
        else { bonoSumado3 = (valorSumado3 - 9) / 2; }
        numRaza3.text = Personaje.Valor(Personaje.ElemRaza("valor3")) + " > " +
                        valorSumado3;
        bonoRaza3.text = bonoAct3 + " > " + bonoSumado3;

        menuSubNiv1.SetActive(true);
    }
```

- **SubirNivel2()**: Función usada para preparar el segundo paso, con el mismo funcionamiento que la anterior función. Se encarga de actualizar todos los valores del menú de elección del valor extra a los que tiene el personaje en el momento de subir de nivel.

```
    public void SubirNivel2()
    {
        //Cambiar nombres, numeros y bonos de cada valor
        int novalorAct1 = Personaje.Valor("FUERZA");
        int nobonoAct1 = Personaje.BonoValor("FUERZA");
        int novalorSumado1 = novalorAct1 + 1;
        int nobonoSumado1;
        if (novalorSumado1 >= 8) { nobonoSumado1 = (novalorSumado1 - 8) / 2; }
        else { nobonoSumado1 = (novalorSumado1 - 9) / 2; }
        numnoclase1.text = Personaje.Valor("FUERZA") + " > " + novalorSumado1;
        bononoclase1.text = nobonoAct1 + " > " + nobonoSumado1;

                                    . . .
```

La función sigue usando esa estructura con el resto de valores. Cuando ya ha preparado todos los valores que muestra la ventana, la activa.

```
        menuSubNiv1.SetActive(false);
        menuSubNiv2.SetActive(true);
    }
```

- **SubirNivel3()**: Función usada para finalizar el proceso de subida de nivel, cerrando las ventanas y actualizando todos los elementos del gestor que pueden ser afectados por el aumento de valores.

```
public void SubirNivel3()
{
    numDados.text = Personaje._otros.numDados.ToString();
    menuSubNiv2.SetActive(false);
    valoresCuerpo.SetActive(false);
    BotonesValores.SetActive(true);
    valoresAlma.SetActive(false);
    Puntos.ActPuntos();
    DesbloqHechiz.DesblHechiz();
    for (int i = 0; i < Personaje.hechizosGuard; i++)
    {
        FuncHechizos.ActHechiz(i);
    }
}
```

- **AbrirValoresCuerpo()**: Función usada para mostrar los valores corporales en la ventana del segundo paso. Simplemente activa el objeto donde se encuentran esos valores y desactiva el objeto de la primera vista de la ventana.

```
void AbrirValoresCuerpo()
{
    BotonesValores.SetActive(false);
    valoresCuerpo.SetActive(true);
}
```

- **AbrirValoresAlma()**: Función usada para mostrar los valores almáticos en la ventana del segundo paso. Simplemente activa el objeto donde se encuentran esos valores y desactiva el objeto de la primera vista de la ventana.

```
void AbrirValoresAlma()
{
    BotonesValores.SetActive(false);
    valoresAlma.SetActive(true);
}
```

- **CerrarValoresCuerpo()**: Función usada para ocultar los valores corporales en la ventana del segundo paso. Simplemente desactiva el objeto donde se encuentran

esos valores y activa el objeto de la primera vista de la ventana.

```
void CerrarValoresCuerpo()
{
    valoresCuerpo.SetActive(false);
    BotonesValores.SetActive(true);
}
```

- **CerrarValoresAlma()**: Función usada para ocultar los valores almáticos en la ventana del segundo paso. Simplemente desactiva el objeto donde se encuentran esos valores y activa el objeto de la primera vista de la ventana.

```
void CerrarValoresAlma()
{
    valoresAlma.SetActive(false);
    BotonesValores.SetActive(true);
}
```

- **ActMenuValores()**: Función usada para actualizar un valor en el bloque de valores del gestor de personaje.

```
void ActMenuValores(string valor)
{
    int valorFinal = Personaje.Valor(valor);
    int bonoFinal = Personaje.BonoValor(valor);
    ValoresMenu.ActValor(valor, valorFinal, bonoFinal);
}
```

6.5.3. Información básica del personaje

Para que el jugador pueda consultar información sobre la clase y la raza a la que pertenece el personaje, se ha implementado un menú donde se pueden encontrar las descripciones de la clase y la raza del personaje. Para crearlo, se ha usado una imágen que sirve de base para crear el menú. En este, para que el jugador pueda decidir qué quiere consultar, se muestran dos botones que permiten mostrar una o otra descripción. Para acceder al menú, se debe pulsar el botón donde se muestra el

nombre, raza y clase del personaje, localizado en la parte superior izquierda del gestor. Ver **Figura 6.5.3/1**.

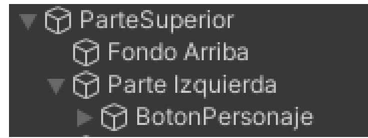

Figura 6.5.3/1: Captura de la jerarquía de la parte superior izquierda del gestor de personajes.

En ese menú se muestran 2 botones que permiten al usuario elegir ver la descripción de la raza o de la clase. Esa descripción se localiza en el objeto “MenuDesc”, y cambia según el botón que se pulse. Ver **Figura 6.5.3/2**.

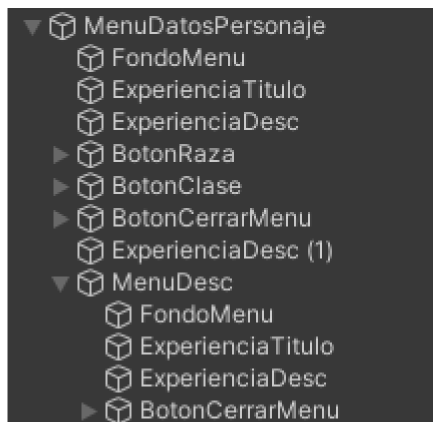

Figura 6.5.3/2: Captura de la jerarquía del menú de datos principales del personaje.

La funcionalidad tanto del botón como la del menú es controlada por el script “Boton Exp”. Ver **Figura 6.5.3/3**.

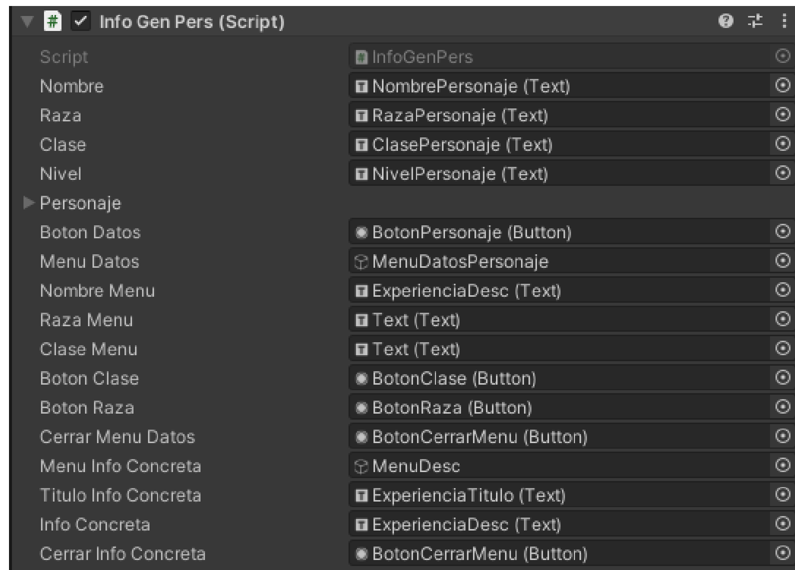

Figura 6.5.3/3: Captura del script en el inspector de Unity y sus atributos.

Sus funciones son:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder manipular los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **Start():** Inicializa y asigna sus funciones para cuando son pulsados a los botones tanto el principal como los del menú. También inicializa los textos del menú y el botón para mostrar la información del personaje nada más cargar la escena.

```
void Start()
{
    botonDatos.onClick.AddListener(AbrirMenu);
    BotonClase.onClick.AddListener(AbrirClase);
    BotonRaza.onClick.AddListener(AbrirRaza);
    CerrarMenuDatos.onClick.AddListener(CerrarMenu);
    CerrarInfoConcreta.onClick.AddListener(CerrarMenuInfoConcreta);


    Nombre.text = Personaje.nombre;
    Raza.text = Personaje.ElemRaza("nombre");
    Clase.text = Personaje.ElemClase("nombre");
    Nivel.text = "Nivel: " + Personaje.nivel;
    NombreMenu.text = "Nombre: " + Personaje.nombre;
    RazaMenu.text = "Raza: " + Personaje._raza.nombreRaza;
    ClaseMenu.text = "Clase: " + Personaje._clase.nombreClase;
}
```

- **AbrirMenu():** Función usada para mostrar el menú cuando se pulsa el botón principal.

```
void AbrirMenu()
{
    MenuDatos.SetActive(true);
}
```

- **AbrirClase():** Función usada para mostrar la descripción de la clase del personaje en el menú.

```
void AbrirClase()
{
    TituloInfoConcreta.text = Personaje._clase.nombreClase;
    InfoConcreta.text = Personaje._clase.descClase;
    MenuInfoConcreta.SetActive(true);
}
```

- **AbrirRaza():** Función usada para mostrar la descripción de la raza del personaje en el menú.

```
void AbrirRaza()
{
    TituloInfoConcreta.text = Personaje._raza.nombreRaza;
    InfoConcreta.text = Personaje._raza.descRaza;
    MenuInfoConcreta.SetActive(true);
}
```

- **CerrarMenu():** Función usada para dejar de mostrar el menú de la información principal del personaje.

```
void CerrarMenu()
{
    MenuDatos.SetActive(false);
}
```

- **CerrarMenuInfoConcreta():** Función usada para dejar de mostrar una descripción y volver a la vista inicial del menú de la información principal del personaje.

```
void CerrarMenuInfoConcreta()
{
    MenuInfoConcreta.SetActive(false);
}
```

6.5.4. Valores

Es el bloque inicial del gestor de personajes, y se ha implementado para que el usuario pueda consultar en todo momento todos los valores del personaje y que pueda manipularlos, implementando varias mecánicas del sistema de rol.

Este bloque se ha separado en 2 partes, una superior, donde se muestran los valores corporales a la izquierda y los valores almáticos a la derecha. Ver **Figura 6.5.4/1**.

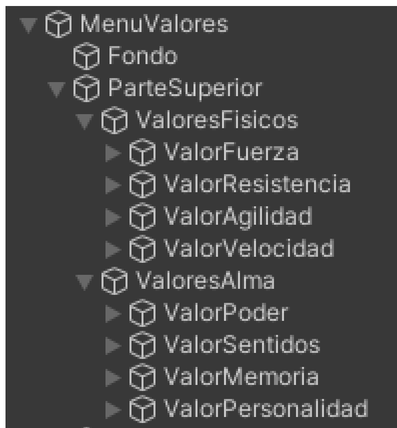

Figura 6.5.4/1: Captura de la jerarquía de la parte superior del bloque de valores.

Cada uno de estos valores es un botón con imágenes en su parte superior e inferior para mostrar el nombre y el valor numérico que representan, En el medio se muestra el bono del valor correspondiente. Ver **Figura 6.5.4/2**.

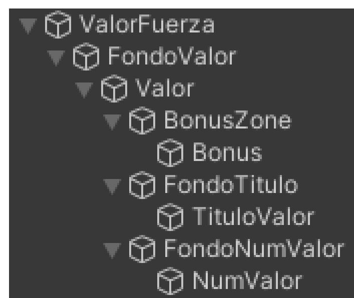

Figura 6.5.4/2: Captura de la jerarquía del valor “Fuerza”.

El usuario puede manipular un valor cualquiera si lo pulsa, lo que mostrará el menú del valor en específico. Existen 2 tipos de menú según si se ha pulsado un valor corporal o uno almático. Ver figuras **6.5.4/3** y **6.5.4/4.**

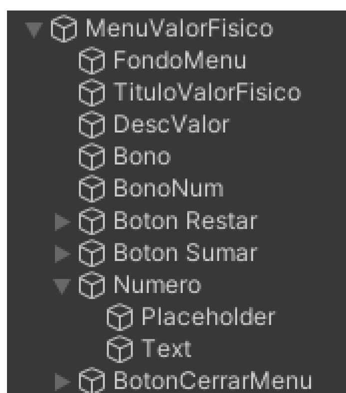

Figura 6.5.4/3: Captura de la jerarquía del menú de un valor corporal.

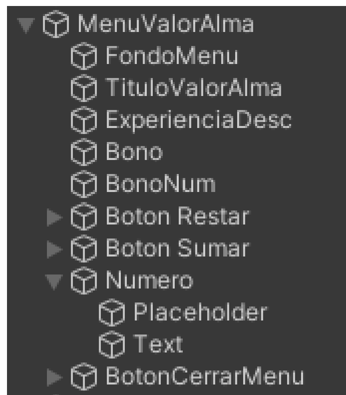

Figura 6.5.4/4: Captura de la jerarquía del menú de un valor almático.

En la parte inferior del bloque se encuentran los valores de “Cuerpo”, “Alma”, “Dados de ataque” y “Acciones por turno”, además de botones que permiten al jugador ejecutar la mecánica de descansos del sistema o reiniciar todos los valores para eliminar todos los efectos de bufos o debufos que tenga el personaje de forma automática. Ver **Figura 6.5.4/5**.

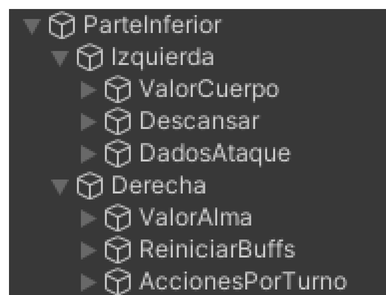

Figura 6.5.4/5: Captura de la jerarquía de la parte inferior del bloque de valores.

Con el fin de permitir consultar y modificar los PC del personaje, se ha implementado un menú al que se accede pulsando el valor “Cuerpo” del bloque. Ver **Figura 6.5.4/6**.

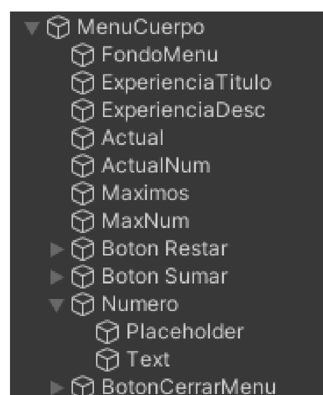

Figura 6.5.4/6: Captura de la jerarquía del menú del valor “Cuerpo”.

De la misma forma, para permitir consultar y modificar los PA del personaje, se ha implementado un menú al que se accede pulsando el valor "Alma" del bloque. Ver **Figura 6.5.4/7**.

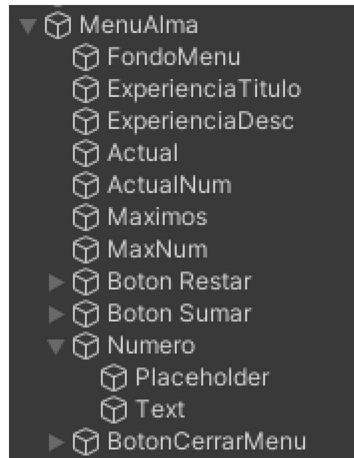

Figura 6.5.4/7: Captura de la jerarquía del menú del valor "Cuerpo".

El usuario también puede consultar el significado del valor "A.P.T", correspondiente a las acciones por turno del personaje, pulsando en él. Haciéndolo se mostrará otro menú con una descripción breve del valor. Ver **Figura 6.5.4/8**.

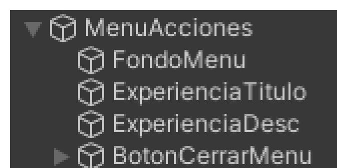

Figura 6.5.4/8: Captura de la jerarquía del menú del valor "Acciones por turno".

El funcionamiento de los valores y sus elementos, a parte de "Cuerpo" y "Alma" es controlado por el script "Valores Menu". Este se encarga de dar funcionalidad a los botones y menús de los valores. Ver **Figura 6.5.4/9A y 6.5.4/9B**.

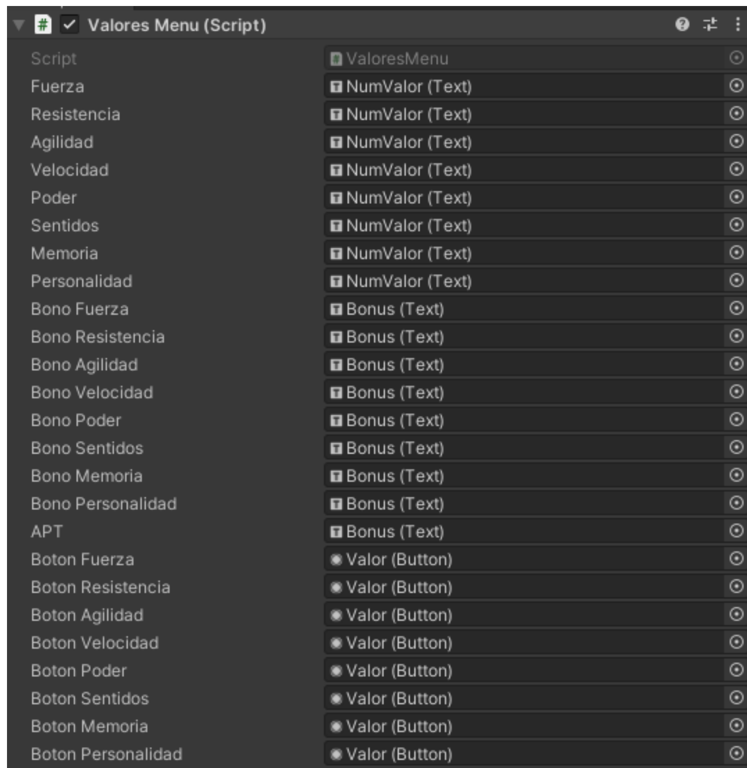

Figura 6.5.4/9A: Captura del script en el inspector de Unity y sus atributos.

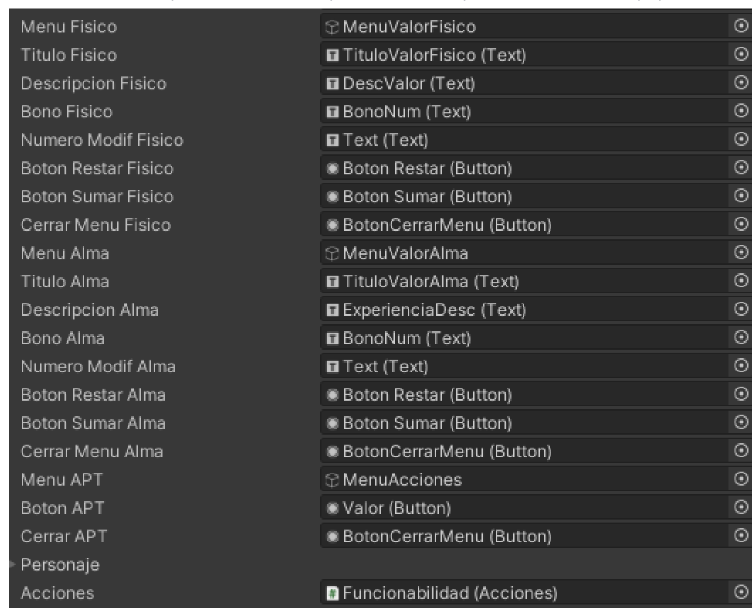

Figura 6.5.4/9B: Captura del script en el inspector de Unity y sus atributos.

Sus funciones son:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder manipular los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **Start():** Inicializa y asigna a los botones de los valores y de sus menús funciones para cuando son pulsados. También inicializa los textos de las acciones ventaja del bloque de acciones y los bonos de cada acción mediante el uso de las funciones “PonerAccionesVent” y “actualizarBonos” del script “Acciones”.

```
void Start()
{
    botonFuerza.onClick.AddListener(Bypass1);
    botonResistencia.onClick.AddListener(Bypass2);
    botonAgilidad.onClick.AddListener(Bypass3);
    botonVelocidad.onClick.AddListener(Bypass4);
    botonPoder.onClick.AddListener(Bypass5);
    botonSentidos.onClick.AddListener(Bypass6);
    botonMemoria.onClick.AddListener(Bypass7);
    botonPersonalidad.onClick.AddListener(Bypass8);
    botonRestarFisico.onClick.AddListener(AplicarDebuffFisico);
    botonSumarFisico.onClick.AddListener(AplicarBuffFisico);
    cerrarMenuFisico.onClick.AddListener(CerrarMenuFisico);
    botonRestarAlma.onClick.AddListener(AplicarDebuffAlma);
    botonSumarAlma.onClick.AddListener(AplicarBuffAlma);
    cerrarMenuAlma.onClick.AddListener(CerrarMenuAlma);
    botonAPT.onClick.AddListener(AbrirAPT);
    cerrarAPT.onClick.AddListener(CerrarAPT);
    Acciones.PonerAccionesVent();
    Acciones.actualizarBonos();
}
```

- **IniciarValores():** Asigna a los valores del bloque su número respectivo guardado en el Personaje actual usando la función “ActValor”.

```
public void IniciarValores()
{
    ActValor("FUERZA", Personaje.Valor("FUERZA"),
             Personaje.BonoValor("FUERZA"));
    ActValor("RESISTENCIA", Personaje.Valor("RESISTENCIA"),
             Personaje.BonoValor("RESISTENCIA"));
    ActValor("AGILIDAD", Personaje.Valor("AGILIDAD"),
             Personaje.BonoValor("AGILIDAD"));
    ActValor("VELOCIDAD", Personaje.Valor("VELOCIDAD"),
             Personaje.BonoValor("VELOCIDAD"));
    ActValor("PODER", Personaje.Valor("PODER"),
             Personaje.BonoValor("PODER"));
    ActValor("SENTIDOS", Personaje.Valor("SENTIDOS"),
             Personaje.BonoValor("SENTIDOS"));
    ActValor("MEMORIA", Personaje.Valor("MEMORIA"),
             Personaje.BonoValor("MEMORIA"));
    ActValor("PERSONALIDAD", Personaje.Valor("PERSONALIDAD"),
             Personaje.BonoValor("PERSONALIDAD"));
}
```

- **Bypass() 1-4:** Conjunto de funciones usadas para preparar el menú de valores corporales. Se ejecuta cuando el botón de un valor corporal es pulsado, y pasan por parámetro a la función "AbrirMenuFisico" los datos del valor al que pertenece el botón pulsado.

```
void Bypass1()
{
    string desc = "Simboliza cómo de fuerte es tu personaje. La habilidad de
                   manipular objetos pesados, golpear con potencia o
                   ejecutar acciones que requieran del uso de la fuerza
                   dependen de este valor.";
    AbrirMenuFisico("FUERZA", desc, bonoFuerza.text);
}
```

- **AbrirMenuFisico():** Función usada para poner los datos del valor seleccionado en el menú de un valor corporal y abrirlo activándolo.

```
public void AbrirMenuFisico(string titulo, string descripcion, string
                            bonoAct)
{
    tituloFisico.text = titulo;
    descripcionFisico.text = descripcion;
    bonoFisico.text = bonoAct;
    menuFisico.SetActive(true);
}
```

- **Bypass() 5-8:** Conjunto de funciones usadas para preparar el menú de valores almáticos. Se ejecuta cuando el botón de un valor almático es pulsado, y pasan por parámetro a la función "AbrirMenuAlma" los datos del valor al que pertenece el botón pulsado.

```
void Bypass5()
{
    string desc = "Simboliza la voluntad del alma que tiene el personaje, la
                   facilidad para ejecutar hechizos, su densidad y su
                   potencia así como el vínculo de sí mismo con el planeta.
                   Los Puntos Almáticos (PA) o por defecto, puntos mágicos o
                   maná estan vinculados a este valor.";
    AbrirMenuAlma("PODER", desc, bonoPoder.text);
}
```

- **AbrirMenuAlma():** Función usada para poner los datos del valor seleccionado en el menú de un valor almático y abrirlo activándolo.

```
public void AbrirMenuAlma(string titulo, string descripcion, string bonoAct)
{
    tituloAlma.text = titulo;
    descripcionAlma.text = descripcion;
    bonoAlma.text = bonoAct;
    menuAlma.SetActive(true);
}
```

- **AplicarDebuffFisico():** Función usada para aplicar un debufo en un valor físico. Actualiza el valor actual en el Personaje y lo que se muestra en el bloque de valores.

```
void AplicarDebuffFisico()
{
    int valorRestar;
    int.TryParse(numeroModifFisico.text, out valorRestar);

    Personaje.ActualizarValorBuff(tituloFisico.text, - valorRestar);
    int valorFinal = Personaje.Valor(tituloFisico.text);
    int bonoFinal = Personaje.BonoValor(tituloFisico.text);
    if (bonoFinal >= 0) { bonoFisico.text = "+" + bonoFinal; }
    else { bonoFisico.text = bonoFinal.ToString(); }
    ActValor(tituloFisico.text, valorFinal, bonoFinal);
}
```

- **AplicarBuffFisico():** Función usada para aplicar un bufo en un valor físico. Actualiza el valor actual en el Personaje y lo que se muestra en el bloque de valores.

```
void AplicarBuffFisico()
{
    int valorSumar;
    int.TryParse(numeroModifFisico.text, out valorSumar);
    Personaje.ActualizarValorBuff(tituloFisico.text, valorSumar);
    int valorFinal = Personaje.Valor(tituloFisico.text);
    int bonoFinal = Personaje.BonoValor(tituloFisico.text);
    if (bonoFinal >= 0) { bonoFisico.text = "+" + bonoFinal; }
    else { bonoFisico.text = bonoFinal.ToString(); }
    ActValor(tituloFisico.text, valorFinal, bonoFinal);
}
```

- **CerrarMenuFisico():** Función usada para cerrar el menú de un valor físico, desactivando el objeto al que pertenece.

```
void CerrarMenuFisico()
{
    menuFisico.SetActive(false);
}
```

- **AplicarDebuffAlma():** Función usada para aplicar un debufo en un valor almático. Actualiza el valor actual en el Personaje y lo que se muestra en el bloque de valores.

```
void AplicarDebuffAlma()
{
    int valorRestar;
    int.TryParse(numeroModifAlma.text, out valorRestar);
    Personaje.ActualizarValorBuff(tituloAlma.text, -valorRestar);
    int valorFinal = Personaje.Valor(tituloAlma.text);
    int bonoFinal = Personaje.BonoValor(tituloAlma.text);
    if (bonoFinal >= 0) { bonoAlma.text = "+" + bonoFinal; }
    else { bonoAlma.text = bonoFinal.ToString(); }
    ActValor(tituloAlma.text, valorFinal, bonoFinal);
}
```

- **AplicarBuffAlma():** Función usada para aplicar un bufo en un valor almático. Actualiza el valor actual en el Personaje y lo que se muestra en el bloque de valores.

```
void AplicarBuffAlma()
{
    int valorSumar;
    int.TryParse(numeroModifAlma.text, out valorSumar);
    Personaje.ActualizarValorBuff(tituloAlma.text, valorSumar);
    int valorFinal = Personaje.Valor(tituloAlma.text);
    int bonoFinal = Personaje.BonoValor(tituloAlma.text);
    if (bonoFinal >= 0) { bonoAlma.text = "+" + bonoFinal; }
    else { bonoAlma.text = bonoFinal.ToString(); }
    ActValor(tituloAlma.text, valorFinal, bonoFinal);
}
```

- **CerrarMenuAlma():** Función usada para cerrar el menú de un valor almático, desactivando el objeto al que pertenece.

```
void CerrarMenuAlma()
{
    menuAlma.SetActive(false);
}
```

- **AbrirAPT():** Función usada para abrir el menú de acciones por turno, activando el objeto al que pertenece.

```
void AbrirAPT()
{
    menuAPT.SetActive(true);
}
```

- **CerrarAPT():** Función usada para cerrar el menú de acciones por turno, desactivando el objeto al que pertenece.

```
void CerrarAPT()
{
    menuAPT.SetActive(false);
}
```

- **ActValor():** Función usada para actualizar lo que un valor del bloque de valores muestra. Comprueba cuál valor actualizar mediante el parámetro de la función "nombreValor".

```
public void ActValor(string nombreValor, int numValor, int bonoValor)
    {
        if (nombreValor == "FUERZA")
        {
            Fuerza.text = numValor.ToString();
            if (bonoValor >= 0) { bonoFuerza.text = "+" + bonoValor; }
            else { bonoFuerza.text = bonoValor.ToString(); }
        }
        else if (nombreValor == "RESISTENCIA")
                                        . . .
```

La función sigue comprobando y actualizando el resto de valores usando la misma estructura mostrada para el valor "Fuerza". Finalmente, llama a la función "actualizarBonos".

```
    Acciones.actualizarBonos();
    }
```

El funcionamiento de los valores "Cuerpo" y "Alma" y sus elementos, además del uso del botón y menú de los descansos es controlado por el script "PCPA". Ver **Figura 6.5.4/10**.

Figura 6.5.4/10: Captura del script en el inspector de Unity y sus atributos.

Sus funciones son:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder manipular los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **Start():** Inicializa y asigna a los botones de los elementos comentados y de sus menús funciones para cuando son pulsados.

```
void Start()
{
    botonCuerpo.onClick.AddListener(AbrirMenuCuerpo);
    botonAlma.onClick.AddListener(AbrirMenuAlma);
    restarPC.onClick.AddListener(RestarPC);
    sumarPC.onClick.AddListener(SumarPC);
    cerrarMenuCuerpo.onClick.AddListener(CerrarCuerpo);
    restarPA.onClick.AddListener(RestarPA);
    sumarPA.onClick.AddListener(SumarPA);
    cerrarMenuAlma.onClick.AddListener(CerrarAlma);
    botonDescanso.onClick.AddListener(AbrirDescanso);
    botonDescansar.onClick.AddListener(CuracionDescanso);
    botonCerrarDescanso.onClick.AddListener(CerrarDescanso);
}
```

- **AbrirMenuCuerpo():** Función usada para poner los datos actuales del valor "Cuerpo" en su menú y abrirlo activándolo.

```
void AbrirMenuCuerpo()
{
    puntosCuerpoActu.text = Personaje.Valor("CUERPOACT").ToString();
    puntosCuerpoMax.text = Personaje.Valor("CUERPOMAX").ToString();
    menuCuerpo.SetActive(true);
}
```

- **AbrirMenuAlma():** Función usada para poner los datos actuales del valor "Alma" en su menú y abrirlo activándolo.

```
void AbrirMenuAlma()
{
    puntosAlmaActu.text = Personaje.Valor("ALMAACT").ToString();
    puntosAlmaMax.text = Personaje.Valor("ALMAMAX").ToString();
    menuAlma.SetActive(true);
}
```

- **RestarPC():** Función usada para restar los PC entrados en el bloque de texto del menú de "Cuerpo" a los PC actuales del personaje. También actualiza los elementos que muestran los PC actuales.

```
void RestarPC()
{
    int valorRestar;
    int.TryParse(puntosEntradosCuerpo.text, out valorRestar);
    int PCAct = Personaje.Valor("CUERPOACT");
    int resultado = PCAct - valorRestar;
    int PCMax = Personaje.Valor("CUERPOMAX");
    puntosCuerpoActu.text = resultado.ToString();
    puntosCuerpo.text = resultado + "/" + PCMax;
    Personaje.ActualizarValor("CUERPOACT", resultado);
}
```

- **SumarPC():** Función usada para sumar los PC entrados en el bloque de texto del menú de “Cuerpo” a los PC actuales del personaje. También actualiza los elementos que muestran los PC actuales.

```
void SumarPC()
{
    int valorSumar;
    int.TryParse(puntosEntradosCuerpo.text, out valorSumar);
    int PCAct = Personaje.Valor("CUERPOACT");
    int resultado = PCAct + valorSumar;
    int PCMax = Personaje.Valor("CUERPOMAX");
    if (resultado >= PCMax) { resultado = PCMax; }
    puntosCuerpoActu.text = resultado.ToString();
    puntosCuerpo.text = resultado + "/" + PCMax;
    Personaje.ActualizarValor("CUERPOACT", resultado);
}
```

- **RestarPA():** Función usada para restar los PA entrados en el bloque de texto del menú de “Alma” a los PA actuales del personaje. También actualiza los elementos que muestran los PA actuales.

```
void RestarPA()
{
    int valorRestar;
    int.TryParse(puntosEntradosAlma.text, out valorRestar);
    int PAAct = Personaje.Valor("ALMAACT");
    int resultado = PAAct - valorRestar;
    int PAMax = Personaje.Valor("ALMAMAX");
    puntosAlmaActu.text = resultado.ToString();
    puntosAlma.text = resultado + "/" + PAMax;
    Personaje.ActualizarValor("ALMAACT", resultado);
```

```
    }
```

- **SumarPA():** Función usada para sumar los PA entrados en el bloque de texto del menú de "Alma" a los PA actuales del personaje. También actualiza los elementos que muestran los PA actuales.

```
    void SumarPA()
    {
        int valorSumar;
        int.TryParse(puntosEntradosAlma.text, out valorSumar);
        int PAAct = Personaje.Valor("ALMAACT");
        int resultado = PAAct + valorSumar;
        int PAMax = Personaje.Valor("ALMAMAX");
        if (resultado >= PAMax) { resultado = PAMax; }
        puntosAlmaActu.text = resultado.ToString();
        puntosAlma.text = resultado + "/" + PAMax;
        Personaje.ActualizarValor("ALMAACT", resultado);
    }
```

- **CerrarCuerpo():** Función usada para cerrar el menú del valor "Cuerpo", desactivando el objeto al que pertenece.

```
    void CerrarCuerpo()
    {
        menuCuerpo.SetActive(false);
    }
```

- **CerrarAlma():** Función usada para cerrar el menú del valor "Alma", desactivando el objeto al que pertenece.

```
    void CerrarAlma()
    {
        menuAlma.SetActive(false);
    }
```

- **ActPuntos():** Función usada para actualizar los elementos que muestran los PC y PA en el bloque de valores. Se usa cuando el personaje sube de nivel, ya que aumentan los PC y PA del personaje.

```
public void ActPuntos()
{
    int PCAct = Personaje.Valor("CUERPOACT");
    int PCMax = Personaje.Valor("CUERPOMAX");
    int PAAct = Personaje.Valor("ALMAACT");
    int PAMax = Personaje.Valor("ALMAMAX");
    float PCHora = Personaje.PuntosHora("CUERPO");
    float PAHora = Personaje.PuntosHora("ALMA");
    puntosCuerpo.text = PCAct + "/" + PCMax;
    puntosAlma.text = PAAct + "/" + PAMax;
    puntosCuerpoHora.text = PCHora.ToString("F1") + " / HORA";
    puntosAlmaHora.text = PAHora.ToString("F1") + " / HORA";
}
```

- **AbrirDescanso():** Función usada para abrir el menú de descanso, activando el objeto al que pertenece.

```
void AbrirDescanso()
{
    menuDescanso.SetActive(true);
}
```

- **CuracionDescanso():** Función usada para realizar un descanso. Calcula los PC y PA que recupera el personaje según el número de horas entradas en el bloque de texto. También se encarga de actualizar los elementos que muestran los PC y PA en el bloque de acciones y de actualizar el personaje con los PC y PA restaurados.

```
void CuracionDescanso()
{
    float valorHoras;
    float.TryParse(horasDescanso.text, out valorHoras);

    float PCHora = Personaje.PuntosHora("CUERPO");
    int PCAct = Personaje.Valor("CUERPOACT");
    int curarCuerpo = (int) (PCHora * valorHoras);
    int resultadoCuerpo = PCAct + curarCuerpo;
    int PCMax = Personaje.Valor("CUERPOMAX");
    if (resultadoCuerpo >= PCMax) { resultadoCuerpo = PCMax; }
    puntosCuerpo.text = resultadoCuerpo + "/" + PCMax;
    Personaje.ActualizarValor("CUERPOACT", resultadoCuerpo);

    float PAHora = Personaje.PuntosHora("ALMA");
```

```
        int PAAct = Personaje.Valor("ALMAACT");
        int curarAlma = (int)(PAHora * valorHoras);
        int resultadoAlma = PAAct + curarAlma;
        int PAMax = Personaje.Valor("ALMAMAX");
        if (resultadoAlma >= PAMax) { resultadoAlma = PAMax; }
        puntosAlma.text = resultadoAlma + "/" + PAMax;
        Personaje.ActualizarValor("ALMAACT", resultadoAlma);
    }
```

- **CerrarDescanso():** Función usada para cerrar el menú de descanso, desactivando el objeto al que pertenece.

```
    void CerrarDescanso()
    {
        menuDescanso.SetActive(false);
    }
```

La funcionalidad del botón para reiniciar los bufos y debufos del personaje la controla el script "Reinicio Buffs". Ver **Figura 6.5.4/11**.

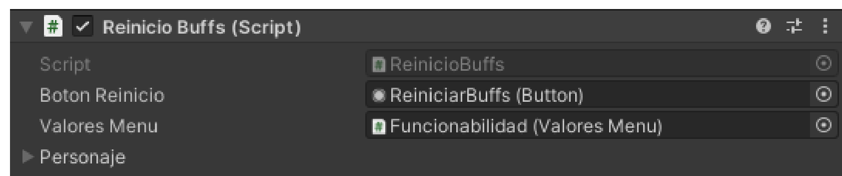

Figura 6.5.4/11: Captura del script en el inspector de Unity y sus atributos.

Sus funciones son:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder manipular los datos del personaje.

```
    public void CargarData(Personaje pers)
    {
        this.Personaje = pers;
    }

    public void GuardarData(Personaje pers)
    {
        pers = this.Personaje;
    }
```

- **Start():** Inicializa y asigna al botón la función "Reiniciar" para cuando es pulsado.

```
void Start()
{
    botonReinicio.onClick.AddListener(Reiniciar);
}
```

- **Reiniciar():** Función encargada de devolver los valores a su valor base mediante la llamada de la función "ReiniciarBuffs". También se encarga de que se actualicen los valores en el bloque de valores llamando la función "IniciarValores".

```
void Reiniciar()
{
    Personaje.ReiniciarBuffs();
    ValoresMenu.IniciarValores();
}
```

6.5.5. Acciones

Es el bloque donde los jugadores pueden hacer las tiradas de todos los tipos de acciones que pueden realizar durante las partidas. Está separado en 3 sub-bloques. Ver **Figura 6.5.5/1**.

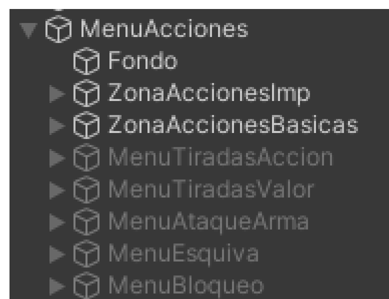

Figura 6.5.5/1: Captura de la jerarquía de del bloque de acciones.

El primero, se encuentra en la parte superior izquierda del bloque, donde se encuentran las acciones ventaja del personaje. De esta forma el usuario puede encontrar las acciones que mejor se le dan al personaje de forma más rápida y cómoda. Ver **Figura 6.5.5/2**.

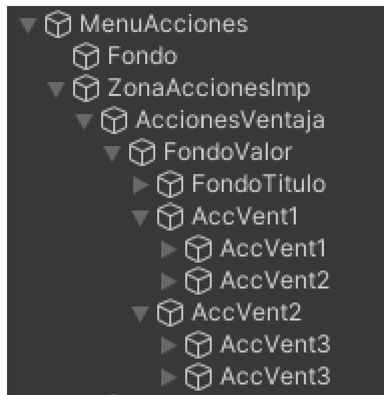

Figura 6.5.5/2: Captura de la jerarquía del sub-bloque de acciones ventaja.

El segundo, se encuentra en la parte derecha del bloque, donde se encuentran todas las acciones disponibles del sistema, incluído un botón para hacer una tirada de un valor por si el personaje quiere ejecutar una acción que no esté especificada. Ver **Figura 6.5.5/3**.

Figura 6.5.5/3: Captura de la jerarquía del sub-bloque de acciones.

Ambos bloques ofrecen acceso al uso de acciones. Cuando se pulsa cualquier acción, para que el personaje sepa en que se basa esa acción, se abre un menú dando información sobre la

acción y con un botón para realizar la tirada de dados pertinente. Al hacerse, se revela el resultado de la tirada. Esto simula el proceso del uso de acciones del sistema Animaia, siguiendo las reglas establecidas. Ver **Figura 6.5.5/4**.

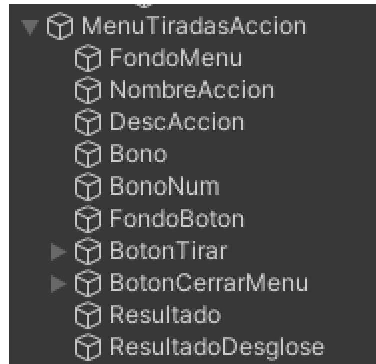

Figura 6.5.5/4: Captura de la jerarquía del menú de una acción.

El funcionamiento de las acciones y sus elementos es controlado por el script "Acciones". Este se encarga de dar funcionalidad a los botones y menús de las acciones. Ver **Figura 6.5.5/5A y 6.5.5/5B**.

Figura 6.5.5/5A: Captura del script en el inspector de Unity y sus atributos.

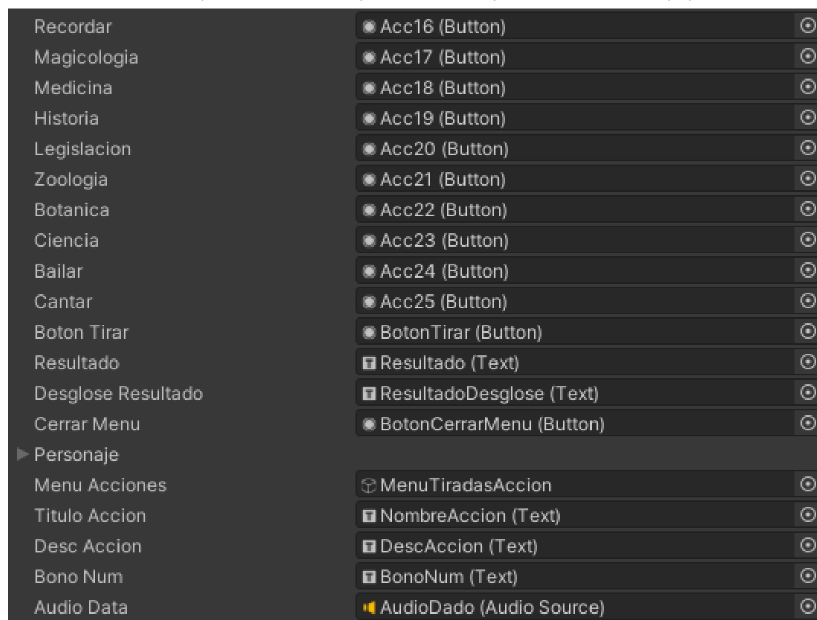

Figura 6.5.5/5B: Captura del script en el inspector de Unity y sus atributos.

Sus atributos son los siguientes:

- **List<string> fuerza:** Listado de las acciones vinculadas al valor “Fuerza”.
- **List<string> resistencia:** Listado de las acciones vinculadas al valor “Resistencia”.
- **List<string> agilidad:** Listado de las acciones vinculadas al valor “Agilidad”.
- **List<string> velocidad:** Listado de las acciones vinculadas al valor “Velocidad”.
- **List<string> poder:** Listado de las acciones vinculadas al valor “Poder”.
- **List<string> sentidos:** Listado de las acciones vinculadas al valor “Sentidos”.
- **List<string> memoria:** Listado de las acciones vinculadas al valor “Memoria”.
- **List<string> personalidad:** Listado de las acciones vinculadas al valor “Personalidad”.
- **bonoFuerza:** Bono vinculado al valor “Fuerza” usado para calcular los resultados de las tiradas de las acciones pertenecientes a ese valor.
- **bonoResistencia:** Bono vinculado al valor “Resistencia” usado para calcular los resultados de las tiradas de las acciones pertenecientes a ese valor.
- **bonoAgilidad:** Bono vinculado al valor “Agilidad” usado para calcular los resultados de las tiradas de las acciones pertenecientes a ese valor.
- **bonoVelocidad:** Bono vinculado al valor “Velocidad” usado para calcular los resultados de las tiradas de las acciones pertenecientes a ese valor.
- **bonoPoder:** Bono vinculado al valor “Poder” usado para calcular los resultados de las tiradas de las acciones pertenecientes a ese valor.
- **bonoSentidos:** Bono vinculado al valor “Sentidos” usado para calcular los resultados de las tiradas de las acciones pertenecientes a ese valor.
- **bonoMemoria:** Bono vinculado al valor “Memoria” usado para calcular los resultados de las tiradas de las acciones pertenecientes a ese valor.
- **bonoPersonalidad:** Bono vinculado al valor “Personalidad” usado para calcular los resultados de las tiradas de las acciones pertenecientes a ese valor.

- **bonoVent1:** Bono vinculado a la primera acción ventaja.
- **bonoVent2:** Bono vinculado a la segunda acción ventaja.
- **bonoVent3:** Bono vinculado a la tercera acción ventaja.
- **bonoVent4:** Bono vinculado a la cuarta acción ventaja.

Sus funciones son las siguientes:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder manipular los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **Start():** Inicializa y asigna a los botones de las acciones y sus menús a sus respectivas funciones para cuando son pulsados. También inicializa los textos del menú de las acciones y se guarda los bonos del personaje en el momento de activar la escena.

```
void Start()
{
    ActivarAccionesVentaja();
    ActivarAccionesFuerza();
    ActivarAccionesResist();
    ActivarAccionesAgilidad();
    ActivarAccionesVelocidad();
    ActivarAccionesPoder();
    ActivarAccionesSentidos();
    ActivarAccionesMemoria();
    ActivarAccionesPersonalidad();
    cerrarMenu.onClick.AddListener(CerrarMenu);
    botonTirar.onClick.AddListener(HacerTirada);
    resultado.enabled = false;
    DesgloseResultado.enabled = false;
    actualizarBonos();
}
```

- **PonerAccionesVent():** Función dedicada a establecer las acciones ventaja del personaje en el sub-bloque correspondiente.

```
public void PonerAccionesVent()
{
    textaccVent1.text = Personaje.accVent1;
    textaccVent2.text = Personaje.accVent2;
    textaccVent3.text = Personaje.accVent3;
    textaccVent4.text = Personaje.accVent4;
}
```

- **actualizarBonos():** Función usada para guardar en el script los bonos de los valores del personaje para cada tipo de acción.

```
public void actualizarBonos()
{
    GuardarBonos();
    GuardarBonosVentaja(textaccVent1.text, 1);
    GuardarBonosVentaja(textaccVent2.text, 2);
    GuardarBonosVentaja(textaccVent3.text, 3);
    GuardarBonosVentaja(textaccVent4.text, 4);
}
```

- **GuardarBonos():** Función usada para guardar en el script los bonos de los valores del personaje en sus respectivos atributos.

```
void GuardarBonos()
{
    bonoFuerza = Personaje.BonoValor("FUERZA");
    bonoResistencia = Personaje.BonoValor("RESISTENCIA");
    bonoAgilidad = Personaje.BonoValor("AGILIDAD");
    bonoVelocidad = Personaje.BonoValor("VELOCIDAD");
    bonoPoder = Personaje.BonoValor("PODER");
    bonoSentidos = Personaje.BonoValor("SENTIDOS");
    bonoMemoria = Personaje.BonoValor("MEMORIA");
    bonoPersonalidad = Personaje.BonoValor("PERSONALIDAD");
}
```

- **GuardarBonosVentaja():** Función usada para guardar en el script los bonos de las acciones ventaja del personaje en sus respectivos atributos. En el siguiente fragmento,

comprueba si la acción pasada por parámetro se encuentra dentro del listado de las acciones vinculadas al valor “Fuerza”. Si es así mira cuál acción ventaja es y le aplica el bono de Fuerza más los puntos extra por ser acción ventaja.

```
void GuardarBonosVentaja(string accion, int num)
{
    if (fuerza.Contains(accion))
    {
        if(num == 1) { bonoVent1 = Personaje.BonoValor("FUERZA") + 2;}
        else if(num == 2) { bonoVent2 = Personaje.BonoValor("FUERZA") + 2;}
        else if (num == 3) { bonoVent3 = Personaje.BonoValor("FUERZA") + 2;}
        else if (num == 4) { bonoVent4 = Personaje.BonoValor("FUERZA") + 2;}
    }
                                . . .
```

Esta comprobación la hace con todos los valores. La función sigue la cadena de *else ifs* con la misma estructura para cada valor.

- **ActivarAccionesVentaja():** Función usada para asignar funciones a los botones de las acciones ventaja.

```
void ActivarAccionesVentaja()
{
    accVent1.onClick.AddListener(AccionVent1);
    accVent2.onClick.AddListener(AccionVent2);
    accVent3.onClick.AddListener(AccionVent3);
    accVent4.onClick.AddListener(AccionVent4);
}
```

- **ActivarAcciones*Valor*():** Grupo de funciones en las que se inicializan y se asignan funciones a los botones correspondientes a las acciones vinculadas al valor específico de la función. También forman los listados de las acciones vinculadas al valor de la función. La parte en rojo del nombre de la función sustituye al nombre de cada valor.

```
void ActivarAccionesFuerza()
{
    saltar.onClick.AddListener(AccionSaltar);
    intimidar.onClick.AddListener(AccionIntimidar);
    fuerza.Add("SALTAR");
    fuerza.Add("INTIMIDAR");
}
```

- **AbrirMenuVent():** Función usada para abrir el menú de una acción ventaja. Comprueba cuál es la acción ventaja pulsada gracias al parámetro de la función "accVent" que contiene el nombre de dicha acción. Dependiendo de esta, llama a la función correspondiente para abrir el menú con la información de dicha función.

```
void AbrirMenuVent(string accVent)
{
    if (accVent == "SALTAR") { AccionSaltar(); }
    else if (accVent == "INTIMIDAR") { AccionIntimidar(); }
    else if (accVent == "NADAR") { AccionNadar(); }
    else if (accVent == "MEDICINA") { AccionMedicina(); }
    else if (accVent == "TREPAR") { AccionTrepar(); }
    else if (accVent == "ROBAR") { AccionRobar(); }
    else if (accVent == "SIGILAR") { AccionSigilar(); }
    else if (accVent == "DISFRAZAR") { AccionDisfrazar(); }
    else if (accVent == "BAILAR") { AccionBailar(); }
    else if (accVent == "CORRER") { AccionCorrer(); }
    else if (accVent == "MAGICOLOGÍA") { AccionMagicologia(); }
    else if (accVent == "PERCIBIR") { AccionPercibir(); }
    else if (accVent == "ESCUCHAR") { AccionEscuchar(); }
    else if (accVent == "DEGUSTAR") { AccionDegustar(); }
    else if (accVent == "OLER") { AccionOler(); }
    else if (accVent == "RECORDAR") { AccionRecordar(); }
    else if (accVent == "HISTORIA") { AccionHistoria(); }
    else if (accVent == "LEGISLACIÓN") { AccionLegislacion(); }
    else if (accVent == "ZOOLOGÍA") { AccionZoologia(); }
    else if (accVent == "BOTÁNICA") { AccionBotanica(); }
    else if (accVent == "CIENCIA") { AccionCiencia(); }
    else if (accVent == "PERSUADIR") { AccionPersuadir(); }
    else if (accVent == "NEGOCIAR") { AccionNegociar(); }
    else if (accVent == "MENTIR") { AccionMentir(); }
    else if (accVent == "CANTAR") { AccionCantar(); }
}
```

- **AccionVent() 1-4:** Grupo de funciones usadas para abrir el menú de una acción ventaja. Se activan al pulsar uno de los botones del sub-bloque de acciones ventaja. Estas

funciones llaman a la función “AbrirMenuVent” pasando por parámetro el nombre de la acción que le corresponde al botón pulsado.

```
void AccionVent1()
{
    AbrirMenuVent(textaccVent1.text);
}
```

- **Accion*Nombre*():** Grupo de funciones usadas para preparar el menú de la acción pulsada con la descripción de dicha acción. Llaman a la función “Ventajaono” pasando por parámetro el nombre de la acción y la descripción. La parte en rojo del nombre de la función sustituye al nombre de cada acción.

```
void AccionSaltar()
{
    string desc = "Vinculada al atributo FUERZA. Simboliza la acción de
                   saltar vertical o longitudinal una cierta distancia para
                   poder superar un obstáculo, alcanzar algo en lo alto o
                   caer sin hacerse daño.";
    Ventajaono(desc, "SALTAR");
}
```

- **Ventajaono():** Función usada para determinar si la acción entrada por parámetro forma parte del conjunto de acciones ventaja o no, y así decidir el bono que se aplica a las tiradas de dados de dicha acción. Cuando comprueba el bono, llama a la función “AbrirMenuAccion”.

```
void Ventajaono(string desc, string action)
{
    if (action == textaccVent1.text) { AbrirMenuAccion(action, desc,
                                                        bonoVent1); }
    else if (action == textaccVent2.text) { AbrirMenuAccion(action, desc,
                                                        bonoVent2); }
    else if (action == textaccVent3.text) { AbrirMenuAccion(action, desc,
                                                        bonoVent3); }
    else if (action == textaccVent4.text) { AbrirMenuAccion(action, desc,
                                                        bonoVent4); }
    else
    {
```

```
            if (fuerza.Contains(action)) { AbrirMenuAccion(action, desc,
                                                bonoFuerza); }
            else if (resistencia.Contains(action)) { AbrirMenuAccion(action,
                                                          desc, bonoResistencia); }
            else if (agilidad.Contains(action)) { AbrirMenuAccion(action, desc,
                                                       bonoAgilidad); }
            else if (velocidad.Contains(action)) { AbrirMenuAccion(action, desc,
                                                        bonoVelocidad); }
            else if (poder.Contains(action)) { AbrirMenuAccion(action, desc,
                                                    bonoPoder); }
            else if (sentidos.Contains(action)) { AbrirMenuAccion(action, desc,
                                                       bonoSentidos); }
            else if (memoria.Contains(action)) { AbrirMenuAccion(action, desc,
                                                      bonoMemoria); }
            else if (personalidad.Contains(action)) { AbrirMenuAccion(action,
                                                           desc, bonoPersonalidad); }
        }
    }
```

- **HacerTirada():** Función con el cometido de calcular la tirada de dados de la acción actual en el menú. Escoge un número de los posibles de un d20 con la función “Random.Range”, suma a ese resultado el bono pertinente y muestra el resultado. Al ejecutar la tirada, esta función activa un sonido usado como feedback simulando el sonido de un dado siendo lanzado en una mesa.

```
    void HacerTirada()
    {
        int min = 1;
        int max = 21;
        int numDadoTirada = Random.Range(min, max);
        int bonoActual;
        int.TryParse(bonoNum.text, out bonoActual);
        int result = bonoActual + numDadoTirada;
        audioData.Play(0);
        if (numDadoTirada == 1)
        {
            resultado.text = "¡Pifia! Acción Fallida";
            DesgloseResultado.text = numDadoTirada + " (d20)";
        }
        else if (numDadoTirada == 20)
        {
            resultado.text = "¡Milagro! Acción Asegurada";
            DesgloseResultado.text = numDadoTirada + " (d20) " + bonoNum.text +
                                     " (Bono " + tituloAccion.text + ")";
```

```
        }
        else
        {
            resultado.text = "Resultado: " + result;
            DesgloseResultado.text = numDadoTirada + " (d20) " + bonoNum.text +
                                     " (Bono " + tituloAccion.text + ")";
        }
        Debug.Log(result);
        resultado.enabled = true;
        DesgloseResultado.enabled = true;
    }
```

- **CerrarMenu():** Función usada para cerrar el menú de la acción, desactivando el objeto al que pertenece.

```
    void CerrarMenu()
    {
        resultado.enabled = false;
        DesgloseResultado.enabled = false;
        menuAcciones.SetActive(false);
    }
```

- **AbrirMenuAccion():** Función usada para abrir el menú de una acción. Añade la información de la acción en los elementos del menú y finalmente activa el objeto del menú para que se visualice.

```
    void AbrirMenuAccion(string nombreAccion, string descripcionAccion, int
                         bono)
    {
        tituloAccion.text = nombreAccion;
        descAccion.text = descripcionAccion;
        if (bono >= 0) { bonoNum.text = "+" + bono; }
        else { bonoNum.text = bono.ToString(); }
        menuAcciones.SetActive(true);
    }
```

Entre todos los botones de acción, existe una excepción, el botón “Valor”. Ese botón, localizado en la esquina inferior derecha del sub-bloque de acciones, sirve para que el usuario pueda hacer tiradas de cualquier valor. Al pulsarlo, abre un menú donde puede seleccionar el valor que quiera, y cuando pulse uno, hará una tirada. El resultado de esa tirada se

muestra en una ventana que desaparece al cabo de 5 segundos. Ver **Figura 6.5.5/6**.

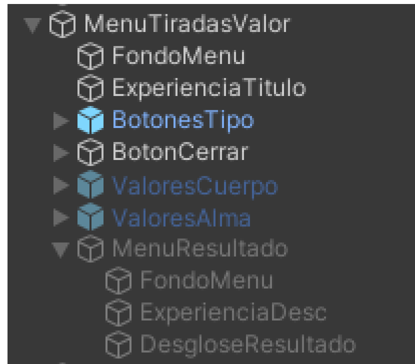

Figura 6.5.5/6: Captura de la jerarquía del menú de tiradas de valor junto el menú del resultado.

La funcionalidad de este menú está controlada por el script "Tiradas Valores". Ver **Figura 6.5.5/7**.

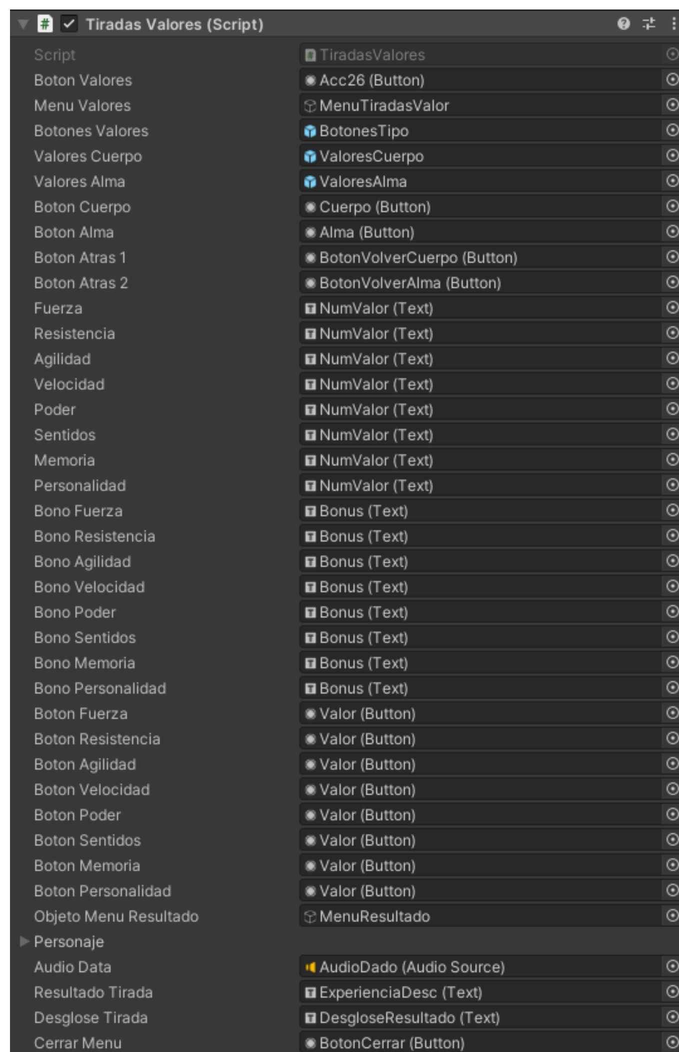

Figura 6.5.5/7: Captura del script en el inspector de Unity y sus atributos.

Sus funciones son:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder manipular los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **Start():** Inicializa y asigna a los botones del menú funciones para cuando son pulsados.

```
void Start()
{
    botonValores.onClick.AddListener(AbrirMenu);
    botonFuerza.onClick.AddListener(UsarFuerza);
    botonResistencia.onClick.AddListener(UsarResistencia);
    botonAgilidad.onClick.AddListener(UsarAgilidad);
    botonVelocidad.onClick.AddListener(UsarVelocidad);
    botonPoder.onClick.AddListener(UsarPoder);
    botonSentidos.onClick.AddListener(UsarSentidos);
    botonMemoria.onClick.AddListener(UsarMemoria);
    botonPersonalidad.onClick.AddListener(UsarPersonalidad);

    botonCuerpo.onClick.AddListener(MostrarValoresCuerpo); ;
    botonAlma.onClick.AddListener(MostrarValoresAlma); ;

    botonAtras1.onClick.AddListener(AtrasCuerpo); ;
    botonAtras2.onClick.AddListener(AtrasAlma); ;

    cerrarMenu.onClick.AddListener(CerrarMenu);
}
```

- **MostrarValoresCuerpo()**: Función usada para mostrar los valores corporales en el menú. Simplemente activa el objeto donde se encuentran esos valores y desactiva el objeto de la primera vista del menú.

```
void MostrarValoresCuerpo()
{
    BotonesValores.SetActive(false);
    ValoresCuerpo.SetActive(true);
}
```

- **MostrarValoresAlma()**: Función usada para mostrar los valores almáticos en el menú. Simplemente, activa el objeto donde se encuentran esos valores y desactiva el objeto de la primera vista del menú.

```
void MostrarValoresAlma()
{
    BotonesValores.SetActive(false);
    ValoresAlma.SetActive(true);
}
```

- **AtrasCuerpo()**: Función usada para ocultar los valores corporales en el menú. Simplemente desactiva el objeto donde se encuentran esos valores y activa el objeto de la primera vista del menú.

```
void AtrasCuerpo()
{
    ValoresCuerpo.SetActive(false);
    BotonesValores.SetActive(true);
}
```

- **AtrasAlma()**: Función usada para ocultar los valores almáticos en el menú. Simplemente, desactiva el objeto donde se encuentran esos valores y activa el objeto de la primera vista del menú.

```
void AtrasAlma()
{
    ValoresAlma.SetActive(false);
    BotonesValores.SetActive(true);
}
```

- **AbrirMenu():** Función usada para abrir el menú de tiradas de valores. Esta llama a la función “IniciarValores”

para que se actualicen todos los valores mostrados en el menú con los valores actuales del personaje y entonces activa el objeto donde se encuentra el menú para que se haga visible.

```
    public void IniciarValores()
    {
        ActValor("FUERZA", Personaje.Valor("FUERZA"),
                 Personaje.BonoValor("FUERZA"));
        ActValor("RESISTENCIA", Personaje.Valor("RESISTENCIA"),
                  Personaje.BonoValor("RESISTENCIA"));
        ActValor("AGILIDAD", Personaje.Valor("AGILIDAD"),
                  Personaje.BonoValor("AGILIDAD"));
        ActValor("VELOCIDAD", Personaje.Valor("VELOCIDAD"),
                  Personaje.BonoValor("VELOCIDAD"));
        ActValor("PODER", Personaje.Valor("PODER"),
                  Personaje.BonoValor("PODER"));
        ActValor("SENTIDOS", Personaje.Valor("SENTIDOS"),
                  Personaje.BonoValor("SENTIDOS"));
        ActValor("MEMORIA", Personaje.Valor("MEMORIA"),
                  Personaje.BonoValor("MEMORIA"));
        ActValor("PERSONALIDAD", Personaje.Valor("PERSONALIDAD"),
                  Personaje.BonoValor("PERSONALIDAD"));
    }
```

- **ActValor():** Función usada para que se actualice un valor en el menú de las tiradas de valor. El valor que actualiza depende del nombre entrado en el parámetro “nombreValor”, y usa el resto de parámetros para modificar el valor en el menú. A continuación solo se muestra la parte de la función que actualiza el valor en caso de que sea “Fuerza”, pero repite este patrón para todos los valores existentes.

```
public void ActValor(string nombreValor, int numValor, int bonoValor)
    {
        if (nombreValor == "FUERZA")
        {
            Fuerza.text = numValor.ToString();
            if (bonoValor >= 0) { bonoFuerza.text = "+" + bonoValor; }
            else { bonoFuerza.text = bonoValor.ToString(); }
        }
        else if (nombreValor == "RESISTENCIA")

                                            . . .
```

- **Usar*Valor*():** Grupo de funciones usadas para hacer la tirada de cada valor en específico. Llama la función "MenúResultado" pasando por parámetro el nombre del valor y el bono del valor. La parte en rojo del nombre de la función sustituye al nombre de cada valor.

```
void UsarFuerza()
   {
       MenuResultado("FUERZA", bonoFuerza.text);
   }
```

- **MenuResultado():** Función con el cometido de calcular la tirada de dados del valor entrado en el parámetro de la función "valor". Escoge un número de los posibles de un d20 con la función "Random.Range", suma a ese resultado el bono pertinente y muestra el resultado. Al ejecutar la tirada, esta función activa un sonido usado como feedback simulando el sonido de un dado siendo lanzado en una mesa. Para mostrar el resultado, usa la co-rutina "StartCooldown".

```
   void MenuResultado(string valor, string bono)
   {
       int min = 1;
       int max = 21;
       int numDadoTirada = Random.Range(min, max);
       int bonoActual;
       int.TryParse(bono, out bonoActual);
       audioData.Play(0);
       int result = bonoActual + numDadoTirada;
       if (numDadoTirada == 1)
       {
           resultadoTirada.text = "¡Pifia! Acción Fallida";
           desgloseTirada.text = numDadoTirada + " (d20)";
       }
       else if (numDadoTirada == 20)
       {
           resultadoTirada.text = "¡Milagro! Acción Asegurada";
           desgloseTirada.text = numDadoTirada + " (d20) " + bono + " (Bono " +
                                 valor + ")";
       }
       else
       {
           resultadoTirada.text = "Resultado: " + result;
```

```
            desgloseTirada.text = numDadoTirada + " (d20) " + bono + " (Bono " +
                                  valor + ")";
        }
        Debug.Log(result);

        StartCoroutine(StartCooldown());
    }
```

- **StartCooldown():** Co-rutina con el cometido de mostrar el resultado de la tirada. Activa la ventana del resultado y pasados 5 segundos, la desactiva.

```
    public IEnumerator StartCooldown()
    {
        ObjetoMenuResultado.SetActive(true);
        yield return new WaitForSeconds(5.0f);
        ObjetoMenuResultado.SetActive(false);
    }
```

- **CerrarMenu():** Función usada para cerrar el menú de los valores, desactivando el objeto al que pertenece.

```
    void CerrarMenu()
    {
        MenuValores.SetActive(false);
    }
```

6.5.6. Ataque, esquiva y bloqueo

En el tercer sub-bloque del bloque de acciones del gestor de personaje es donde se encuentran las acciones de combate. Al ser acciones especiales que solo pueden ocurrir en el contexto de un combate, están separadas del resto de acciones, y se pueden localizar en la parte izquierda del bloque de acciones, por debajo de las acciones ventaja. Ver **Figura 6.5.6/1**.

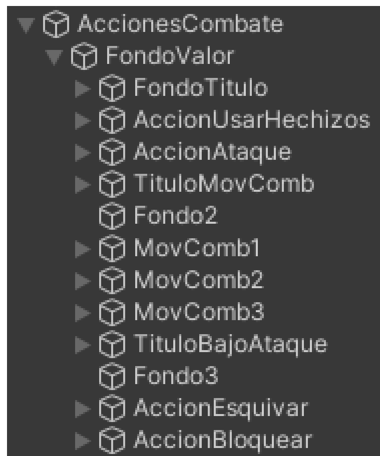

Figura 6.5.6/1: Captura de la jerarquía del sub-bloque de acciones de combate.

El sub-bloque está separado en tres partes diferentes. La primera es donde el usuario puede elegir si atacar usando su arma o usar una habilidad pulsando sus respectivos botones. Al pulsar el botón de ataque, se abre el menú de ataque con el arma e inicia el proceso de una ataque. Ver **Figura 6.5.6/2**.

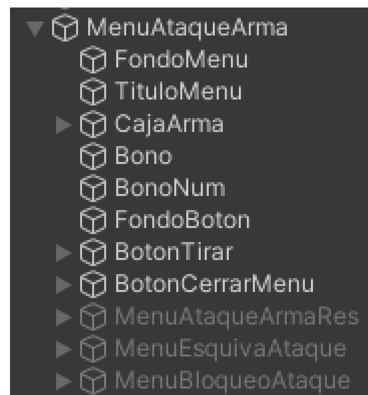

Figura 6.5.6/2: Captura de la jerarquía del menú de ataque con arma.

Para implementar un ataque con arma, se han seguido los pasos que el sistema de rol explica, representando cada uno de ellos por un menú diferente. Primero se muestra al jugador los datos del arma que va a usar, y debe pulsar un botón para ejecutar el ataque. Entonces se calcula la tirada de daño, tal y como el sistema lo indica. Se muestra un menú nuevo con el resultado de la tirada y entonces empieza la fase de esquiva o bloqueo, donde se le pregunta al jugador si el enemigo hace una de las dos acciones que ha tenido que declarar previamente. El menú del resultado tiene 2 botones para responder a esa pregunta. **Figura 6.5.6/3**.

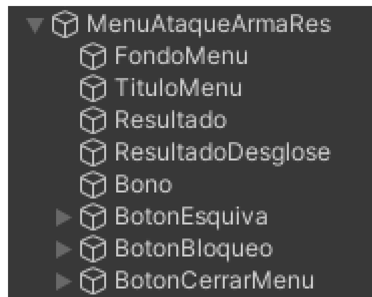

Figura 6.5.6/3: Captura de la jerarquía del menú del resultado del ataque con arma.

Al pulsar uno de esos botones, se calcula la tirada respectiva y se muestra en otro menú diferente su resultado y una caja de texto donde el jugador debe introducir el resultado de la acción del enemigo para resolver el resultado final del ataque, finalizando así el proceso. Ver **Figuras 6.5.6/4**.

Figura 6.5.6/4: A la izquierda, captura de la jerarquía del menú del resultado de la esquiva. A la derecha, captura de la jerarquía del menú del resultado del bloqueo.

Pulsar el botón correspondiente de usar habilidades lleva al usuario al bloque de habilidades del gestor, que se comentará en el punto 6.5.7.

La funcionalidad de ambos botones y del menú de ataque con arma se encuentran en el script “Combat Attacks”. Ver **Figura 6.5.6/5**.

Figura 6.5.6/5: Captura del script y sus atributos.

Sus funciones son:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder manipular los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **Start():** Inicializa y asigna a los botones de las acciones de combate y de sus menús funciones para cuando son pulsados, llamando la función "IniciarBotones" en el primer farme al activarse el gestor.

```
void Start()
{
    IniciarBotones();
}
```

- **IniciarBotones():** Inicializa y asigna a los botones de las acciones de combate y de sus menús funciones para cuando son pulsados.

```
void IniciarBotones()
{
    botonAtacar.onClick.AddListener(atacarArma);
    botonHechizos.onClick.AddListener(usarHechizo);
    botonAtacarMenuAtacar.onClick.AddListener(Atacar);
    botonEsquivarAtaque.onClick.AddListener(Esquiva);
    botonBloquearAtaque.onClick.AddListener(Bloqueo);
    botonCalcularEsquivaFinal.onClick.AddListener(CalcularEsquiva);
    botonCalcularBloqueoFinal.onClick.AddListener(CalcularBloqueo);
    botonCerrarAtaque.onClick.AddListener(CerrarAtaque);
    botonCerrarResuAtaque.onClick.AddListener(CerrarResu);
    botonCerrarEsquiva.onClick.AddListener(CerrarEsquiva);
    botonCerrarBloqueo.onClick.AddListener(CerrarBloqueo);
}
```

- **atacarArma():** Función usada cuando se pulsa el botón de atacar con el arma del sub-bloque. Prepara el menú de ataque con la información del arma del personaje y lo activa.

```
void atacarArma()
{
    nombreArma.text = Personaje.ArmaEquipada.nombreArma;
    tipoArma.text = Personaje.ArmaEquipada.tipoArma;
    dadoArma.text = "d" + Personaje.ArmaEquipada.dadoArma;
    numDados.text = Personaje._otros.numDados.ToString();
    MenuAtaqueArma.SetActive(true);
}
```

- **Atacar():** Función usada cuando se pulsa el botón de atacar en el menú de ataque, ejecutando la acción e iniciando el proceso de ataque. Calcula la tirada de dados del ataque según el tipo de dados que use el arma y el número de dados de ataque que dispone el personaje. También tiene en cuenta si el personaje ataca habiendo usado algún hechizo que afecte el ataque previamente. Finalmente prepara el texto del menú del resultado de la tirada y muestra el menú para que el usuario pase al siguiente paso del ataque.

```
void Atacar()
{
    int min = 1;
    int max = Personaje.ArmaEquipada.dadoArma + 1;
    int numDadoTirada;
    int result = 0;
    audioData.Play(0);
    int bonoActual = Personaje.BonoValor(Personaje.ArmaEquipada.tipoArma);
    for (int i = 0; i < Personaje._otros.numDados; i++)
    {
        numDadoTirada = Random.Range(min, max);
        result = result + numDadoTirada;
        desgloseTirada.text += numDadoTirada + " (" + dadoArma.text + ") " +
                               "+ ";
    }
    result = result + bonoActual;

    if (Personaje.cargado)
    {
        result = result + Personaje.valorCargado;
        desgloseTirada.text += bonoActual + " (Bono " +
                               Personaje.ArmaEquipada.tipoArma + ") + " +
                               Personaje.valorCargado + " (Ataque cargado)";
        Personaje.valorCargado = 0;
        Personaje.cargado = false;
    }
    else
    {
        desgloseTirada.text += bonoActual + " (Bono " + tipoArma.text + ")";
    }
    resultadoAtaqueArma = result;
    resultadoTirada.text = "Resultado: " + result;

    Debug.Log(result);
    MenuAtaqueArmaRes.SetActive(true);
}
```

- **Esquiva():** Función usada cuando se pulsa el botón de esquiva en el menú del resultado de daño del ataque, ejecutando la acción de esquiva. Calcula la tirada de dados de la esquiva usando el bono de agilidad del personaje. Finalmente prepara el texto del menú del resultado de la tirada y muestra el menú final.

```
void Esquiva()
{
    int min = 1;
    int max = 21;
    int numDadoTirada = Random.Range(min, max);
    int bonoActual = Personaje.BonoValor("AGILIDAD");
    int result;
    audioData.Play(0);
    if (numDadoTirada == 1)
    {
        resultadoTiradaEsquiva.text = "¡Pifia!";
        desgloseTiradaEsquiva.text = numDadoTirada + " (d20)";
        esquivaAtacante = "Pifia";
    }
    else if (numDadoTirada == 20)
    {
        resultadoTiradaEsquiva.text = "¡Milagro!";
        desgloseTiradaEsquiva.text = numDadoTirada + " (d20) ";
        esquivaAtacante = "Milagro";
    }
    else
    {
        result = numDadoTirada + bonoActual;
        esquivaAtacante = result.ToString();
        resultadoTiradaEsquiva.text = "Tu resultado: " + result;
        desgloseTiradaEsquiva.text = numDadoTirada + " (d20) + " +
                                     bonoActual + " (Bono AGILIDAD)";
        Debug.Log(result);
    }

    MenuEsquivaAtaque.SetActive(true);
}
```

- **Bloqueo():** Función usada cuando se pulsa el botón de bloqueo en el menú del resultado de daño del ataque, ejecutando la acción de bloqueo. Calcula la tirada de dados del bloqueo usando el bono vinculado al arma del personaje. Finalmente prepara el texto del menú del resultado de la tirada y muestra el menú final.

```
    void Bloqueo()
    {
        int min = 1;
        int max = 21;
        int numDadoTirada = Random.Range(min, max);
        int bonoActual = Personaje.BonoValor(Personaje.ArmaEquipada.tipoArma);
        int result;
        audioData.Play(0);
        if (numDadoTirada == 1)
        {
            resultadoTiradaBloqueo.text = "¡Pifia!";
            desgloseTiradaBloqueo.text = numDadoTirada + " (d20)";
            bloqueoAtacante = "Pifia";
        }
        else if (numDadoTirada == 20)
        {
            resultadoTiradaBloqueo.text = "¡Milagro!";
            desgloseTiradaBloqueo.text = numDadoTirada + " (d20) ";
            bloqueoAtacante = "Milagro";
        }
        else
        {
            result = numDadoTirada + bonoActual;
            bloqueoAtacante = result.ToString();
            resultadoTiradaBloqueo.text = "Tu resultado: " + result;
            desgloseTiradaBloqueo.text = numDadoTirada + " (d20) + " +
                                         bonoActual + " (Bono " + tipoArma.text
                                         + ") ";
            Debug.Log(result);
        }

        MenuBloqueoAtaque.SetActive(true);
    }
```

- **CalcularEsquiva():** Función usada para decidir el resultado final del ataque teniendo en cuenta el resultado de esquiva del atacante y el resultado de esquiva del atacado entrado en el bloque de texto del menú. Compara ambos resultados y muestra el desenlace final del ataque según lo establecido en el sistema de rol.

```
void CalcularEsquiva()
    {
        int valorEsquivaEnemiga;
        int valorEsquivaAtacante;
        int dmgTotal;
```

```
        if(resultadoEsquivaEnemiga.text == "Pifia")
        {
            if(esquivaAtacante == "Pifia")
            {
                resultadoFinalEsquiva.text = "Ataque fallido";
            }
            else
            {
                dmgTotal = resultadoAtaqueArma * 2;
                resultadoFinalEsquiva.text = "Infliges " + dmgTotal + " de
                                              daño";
            }
        }
        else if (resultadoEsquivaEnemiga.text == "Milagro")
        {
            if (esquivaAtacante == "Milagro")
            {
                dmgTotal = resultadoAtaqueArma;
                resultadoFinalEsquiva.text = "Infliges " + dmgTotal + " de
                                              daño";
            }
            else
            {
                resultadoFinalEsquiva.text = "¡El enemigo contraataca!";
            }
        }
        else
        {
            if (esquivaAtacante == "Pifia")
            {
                resultadoFinalEsquiva.text = "¡El enemigo contraataca!";
            }
            else if (esquivaAtacante == "Milagro" || Personaje.oculto)
            {
                dmgTotal = resultadoAtaqueArma * 2;
                resultadoFinalEsquiva.text = "¡Golpe Crítico! Infliges " +
                                              dmgTotal + " de daño";
                Personaje.oculto = false;
            }
            else
            {
                int.TryParse(resultadoEsquivaEnemiga.text, out
                             valorEsquivaEnemiga);
                int.TryParse(esquivaAtacante, out valorEsquivaAtacante);
                if(valorEsquivaEnemiga >= valorEsquivaAtacante)
                {
                    resultadoFinalEsquiva.text = "¡El enemigo esquivó el
                                                  ataque!";
                }
                else
```

```
                {
                    dmgTotal = resultadoAtaqueArma;
                    resultadoFinalEsquiva.text = "Infliges " + dmgTotal + " de
                                                   daño";
                }
            }
        }
    }
```

- **CalcularEsquiva():** Función usada para decidir el resultado final del ataque teniendo en cuenta el resultado de bloqueo del atacante y el resultado de bloqueo del atacado entrado en el bloque de texto del menú. Compara ambos resultados y muestra el desenlace final del ataque según lo establecido en el sistema de rol.

```
void CalcularBloqueo()
    {
        int valorBloqueoEnemiga;
        int valorBloqueoAtacante;
        int dmgTotal;
        if (resultadoBloqueoEnemigo.text == "Pifia")
        {
            if (bloqueoAtacante == "Pifia")
            {
                resultadoFinalEsquiva.text = "Ataque fallido";
            }
            else
            {
                dmgTotal = resultadoAtaqueArma * 2;
                resultadoFinalBloqueo.text = "Infliges " + dmgTotal + " de
                                               daño";
            }
        }
        else if (resultadoBloqueoEnemigo.text == "Milagro")
        {
            if (bloqueoAtacante == "Milagro")
            {
                dmgTotal = resultadoAtaqueArma;
                resultadoFinalBloqueo.text = "Infliges " + dmgTotal + " de
                                               daño";
            }
            else
            {
                resultadoFinalBloqueo.text = "¡El enemigo contraataca!";
            }
```

```
        }
        else
        {
            if (bloqueoAtacante == "Pifia")
            {
                resultadoFinalBloqueo.text = "¡El enemigo contraataca!";
            }
            else if (bloqueoAtacante == "Milagro" || Personaje.oculto)
            {
                dmgTotal = resultadoAtaqueArma * 2;
                resultadoFinalBloqueo.text = "¡Golpe Crítico! Infliges " +
                                              dmgTotal + " de daño";
                Personaje.oculto = false;
            }
            else
            {
                int.TryParse(resultadoBloqueoEnemigo.text, out
                             valorBloqueoEnemiga);
                int.TryParse(bloqueoAtacante, out valorBloqueoAtacante);
                if (valorBloqueoEnemiga >= valorBloqueoAtacante)
                {
                    dmgTotal = resultadoAtaqueArma - (valorBloqueoEnemiga -
                                                      valorBloqueoAtacante);
                    if (dmgTotal <= 0)
                    {
                        dmgTotal = 0;
                        resultadoFinalBloqueo.text = "¡El enemigo bloqueó el
                                                      ataque completamente!";
                    }
                    else
                    {
                        resultadoFinalBloqueo.text = "¡El enemigo se defendió!
                                                      Infliges " + dmgTotal + "
                                                      de daño";
                    }
                }
                else
                {
                    dmgTotal = resultadoAtaqueArma;
                    resultadoFinalBloqueo.text = "Infliges " + dmgTotal + " de
                                                  daño";
                }
            }
        }
    }
```

- **CerrarAtaque():** Función usada para cerrar el menú de ataque, desactivando el objeto al que pertenece.

```
void CerrarAtaque()
{
    MenuAtaqueArma.SetActive(false);
}
```

- **CerrarResu():** Función usada para cerrar el menú del resultado de daño del ataque, desactivando el objeto al que pertenece.

```
void CerrarResu()
{
    MenuAtaqueArmaRes.SetActive(false);
    desgloseTirada.text = " ";
}
```

- **CerrarEsquiva():** Función usada para cerrar el menú del resultado de la esquiva en ataque, desactivando el objeto al que pertenece.

```
void CerrarEsquiva()
{
    resultadoFinalEsquiva.text = " ";
    MenuEsquivaAtaque.SetActive(false);
}
```

- **CerrarBloqueo():** Función usada para cerrar el menú del resultado del bloqueo en ataque, desactivando el objeto al que pertenece.

```
void CerrarBloqueo()
{
    resultadoFinalBloqueo.text = " ";
    MenuBloqueoAtaque.SetActive(false);
}
```

- **usarHechizo():** Función usada cuando se pulsa el botón para usar una habilidad en el sub-bloque. Simplemente muestra el bloque de habilidades del gestor.

```
void usarHechizo()
{
    BotonDesplegable.text = "HABILIDADES";
    MenuHechizos.SetActive(true);
    MenuAcciones.SetActive(false);
}
```

En el sub-bloque también existen botones de las acciones de movimiento enfrentarse, alejarse y proteger, pero sirven para que el jugador recuerde que puede usarlas, no tienen funcionalidad.

Finalmente están los botones correspondientes a las acciones de esquiva y bloqueo, disponibles para que el jugador pueda ejecutarlas en el momento de ser atacado. Al pulsarlos, muestran un menú con el resultado de la tirada de dados de la acción. Ver **Figura 6.5.6/6**.

Figura 6.5.6/4: A la izquierda, captura de la jerarquía del menú del resultado de la esquiva. A la derecha, captura de la jerarquía del menú del resultado del bloqueo.

Su funcionamiento, tanto el de los botones como el de los menús, está controlado por el script “Esquiva Bloqueo”. Ver **Figura 6.5.6/7**.

Figura 6.5.6/5: Captura del script y sus atributos.

Sus funciones son:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder manipular los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **Start():** Inicializa y asigna a los botones de las acciones de combate y de sus menús funciones para cuando son pulsados en el primer frame al activarse el gestor.

```
void Start()
{
    botonEsquivarAtaque.onClick.AddListener(Esquiva);
    botonBloquearAtaque.onClick.AddListener(Bloqueo);
    botonCerrarEsquiva.onClick.AddListener(CerrarEsquiva);
    botonCerrarBloqueo.onClick.AddListener(CerrarBloqueo);
}
```

- **Esquiva():** Función usada cuando se pulsa el botón de esquiva en sub-bloque, ejecutando la acción de esquiva. Calcula la tirada de dados de la esquiva usando el bono de agilidad del personaje. Finalmente prepara el texto del menú del resultado de la tirada y muestra el menú del resultado.

```
void Esquiva()
{
    int min = 1;
    int max = 21;
    int numDadoTirada = Random.Range(min, max);
    int bonoActual = Personaje.BonoValor("AGILIDAD");
    audioData.Play(0);
    int result;
    if (numDadoTirada == 1)
    {
        resultadoTiradaEsquiva.text = "¡Pifia!";
        desgloseTiradaEsquiva.text = numDadoTirada + " (d20)";
    }
    else if (numDadoTirada == 20)
    {
        resultadoTiradaEsquiva.text = "¡Milagro!";
        desgloseTiradaEsquiva.text = numDadoTirada + " (d20) ";
    }
    else
    {
        result = numDadoTirada + bonoActual;
        resultadoTiradaEsquiva.text = "Tu resultado: " + result;
        desgloseTiradaEsquiva.text = numDadoTirada + " (d20) + " +
                                     bonoActual + " (Bono AGILIDAD)";
        Debug.Log(result);
    }
    MenuEsquivaAtaque.SetActive(true);
}
```

- **Bloqueo():** Función usada cuando se pulsa el botón de bloqueo en sub-bloque, ejecutando la acción de esquiva. Calcula la tirada de dados de la esquiva usando el bono de resistencia del personaje. Finalmente prepara el texto del menú del resultado de la tirada y muestra el menú del resultado.

```
void Bloqueo()
{
    int min = 1;
    int max = 21;
    int numDadoTirada = Random.Range(min, max);
    int bonoActual = Personaje.BonoValor("RESISTENCIA");
    audioData.Play(0);
    int result;
    if (numDadoTirada == 1)
    {
        resultadoTiradaBloqueo.text = "¡Pifia!";
        desgloseTiradaBloqueo.text = numDadoTirada + " (d20)";
    }
    else if (numDadoTirada == 20)
    {
        resultadoTiradaBloqueo.text = "¡Milagro!";
        desgloseTiradaBloqueo.text = numDadoTirada + " (d20) ";
    }
    else
    {
        result = numDadoTirada + bonoActual;
        resultadoTiradaBloqueo.text = "Tu resultado: " + result;
        desgloseTiradaBloqueo.text = numDadoTirada + " (d20) + " +
                                     bonoActual + " (Bono RESISTENCIA)";
        Debug.Log(result);
    }
    MenuBloqueoAtaque.SetActive(true);
}
```

- **CerrarEsquiva():** Función usada para cerrar el menú del resultado de la esquiva, desactivando el objeto al que pertenece.

```
void CerrarEsquiva()
{
    MenuEsquivaAtaque.SetActive(false);
}
```

- **CerrarBloqueo():** Función usada para cerrar el menú del resultado del bloqueo, desactivando el objeto al que pertenece.

```
void CerrarBloqueo()
{
    MenuBloqueoAtaque.SetActive(false);
}
```

6.5.7. Habilidades

Es el bloque donde los jugadores pueden consultar las habilidades del personaje, usarlas y aprenderlas. El bloque consiste del listado de habilidades del personaje. Como el sistema lo indica, el personaje puede aprender hasta ocho habilidades, que es el número de huecos que presenta el listado del bloque. Ver **Figura 6.5.7/1**.

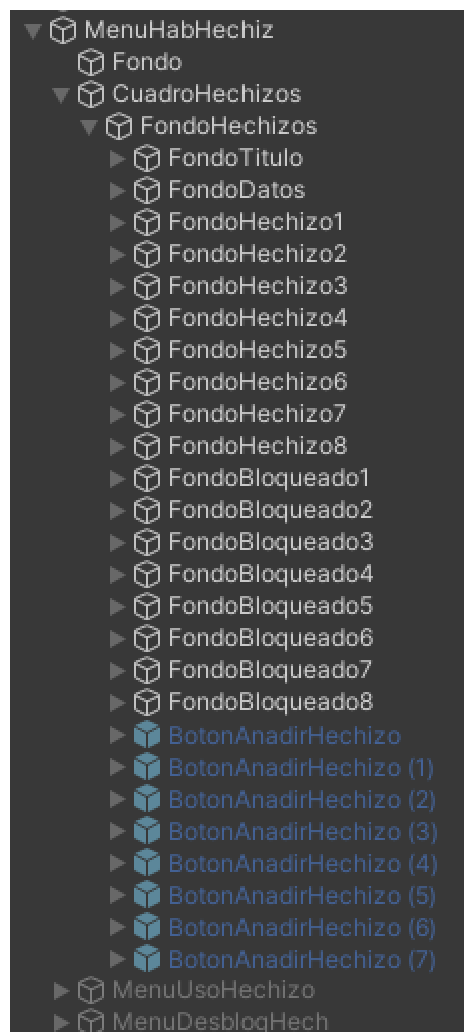

Figura 6.5.7/1: Captura de la jerarquía del bloque de habilidades.

A cada hueco se muestra uno de los tres tipos diferentes de elementos dependiendo del estado del personaje conforme su valor base de "Memoria". Estos pueden ser:

- **Bloqueado:** Cuando el valor de "Memoria" es inferior al que se necesita para asignar una habilidad en ese hueco,

mostrará un mensaje mostrando el valor de "Memoria" necesario.

- **Por aprender:** Cuando el valor de "Memoria" es igual o superior al que se necesita para asignar una habilidad en ese hueco, mostrará un botón con un mensaje conforme es posible la asignación de una habilidad.
- **Aprendido:** Cuando el personaje ya ha asignado una habilidad en ese hueco, se mostrará información de la habilidad en concreto. Ver **Figura 6.5.7/2**.

Figura 6.5.7/2: Captura de la jerarquía de un hueco con la habilidad aprendida.

Para que el programa sepa cuales son los huecos bloqueados y cuáles no lo están, se usa el script "Desbloq Hechiz". Ver **Figura 6.5.7/3**.

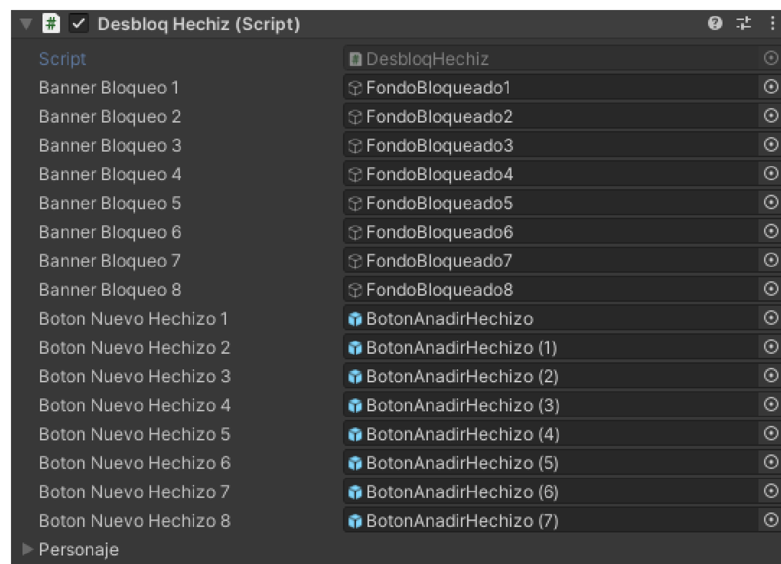

Figura 6.5.6/5: Captura del script y sus atributos.

Este script solo contiene 2 funciones, que son:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder acceder a los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **DesblHechiz():** Función usada para comprobar cuáles son los huecos bloqueados o desbloqueados según las habilidades aprendidas por el personaje y su valor base de "Memoria". Esta función se ejecuta al inicarse la escena y al subir de nivel por si se ha cambiado el valor "Memoria".

```
public void DesblHechiz()
{

    int memoPers = Personaje.Valor("MEMORIA");
    Debug.Log(memoPers);
    int hechiActPers = Personaje.hechizosGuard;
    Debug.Log(hechiActPers);
    if (memoPers >= 6)
    {
        bannerBloqueo1.SetActive(false);
        if (hechiActPers == 0)
        {
            botonNuevoHechizo1.SetActive(true);
        }
        if (memoPers >= 8)
        {
            bannerBloqueo2.SetActive(false);
            if (hechiActPers <= 1)
            {
                botonNuevoHechizo2.SetActive(true);
            }
            if (memoPers >= 10)
            {
                bannerBloqueo3.SetActive(false);
                if (hechiActPers <= 2)
                {
                    botonNuevoHechizo3.SetActive(true);
                }
                if (memoPers >= 12)
                {
```

```
                    bannerBloqueo4.SetActive(false);
                    if (hechiActPers <= 3)
                    {
                        botonNuevoHechizo4.SetActive(true);
                    }
                    if (memoPers >= 14)
                    {
                        bannerBloqueo5.SetActive(false);
                        if (hechiActPers <= 4)
                        {
                            botonNuevoHechizo5.SetActive(true);
                        }
                        if (memoPers >= 16)
                        {
                            bannerBloqueo6.SetActive(false);
                            if (hechiActPers <= 5)
                            {
                                botonNuevoHechizo6.SetActive(true);
                            }
                            if (memoPers >= 18)
                            {
                                bannerBloqueo7.SetActive(false);
                                if (hechiActPers <= 6)
                                {
                                    botonNuevoHechizo7.SetActive(true);
                                }
                                if (memoPers >= 20)
                                {
                                    bannerBloqueo8.SetActive(false);
                                    if (hechiActPers <= 7)
                                    {
                                        botonNuevoHechizo8.SetActive(true);
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Para que el jugador pueda asignar habilidades al personaje, cuando se desbloquea un hueco de habilidad, se muestra un botón para que, cuando el jugador lo pulse, se inicie el proceso de aprendizaje de una habilidad nueva. Cuando el jugador lo pulsa, se muestra un menú con botones representando todas

las habilidades que el personaje puede aprender. Ver **Figura 6.5.7/4**.

Figura 6.5.7/4: Captura de la jerarquía del menú de habilidades por aprender.

Cuando el personaje pulsa una de las habilidades, se muestra un menú con la información de la habilidad seleccionada. En este menú es dónde el jugador puede confirmar la selección de la nueva habilidad pulsando el botón llamado “BotonTirar”. Ver **Figura 6.5.7/5**.

Figura 6.5.7/5: Captura de la jerarquía del menú de una habilidad.

El script encargado de controlar todo el proceso de asignación de nuevas habilidades es el “Nuevo Hechizo”. **Figura 6.5.7/6A y 6.5.7/6B**.

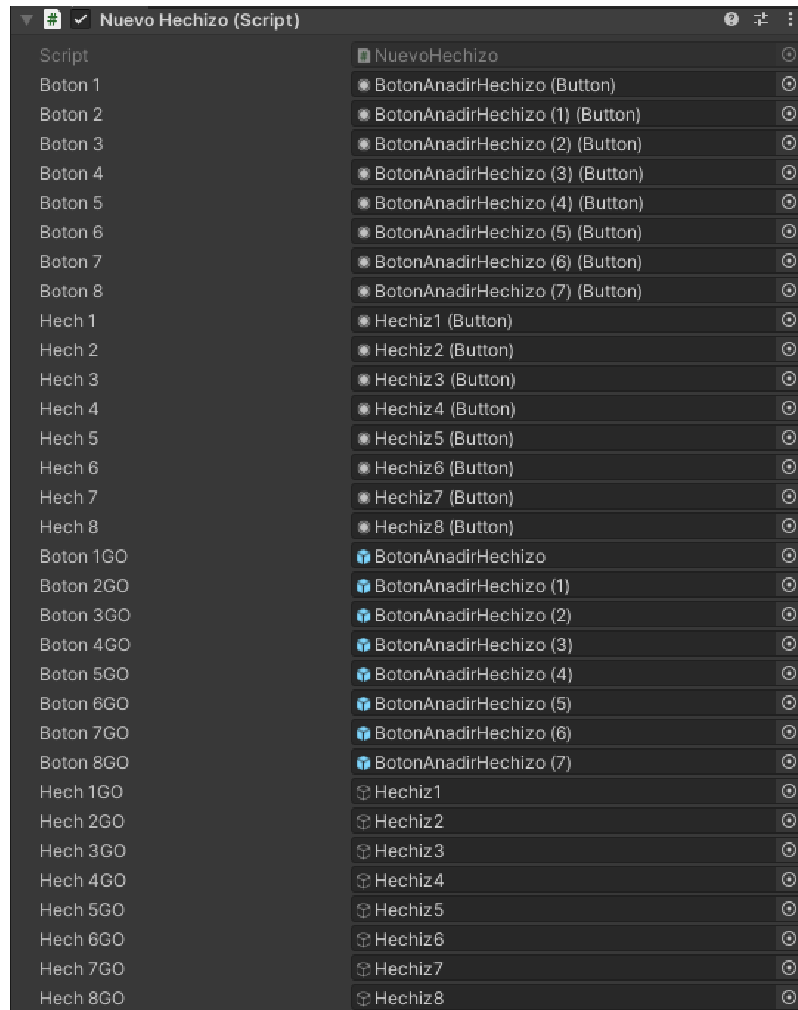

Figura 6.5.7/6A: Captura del script y sus atributos.

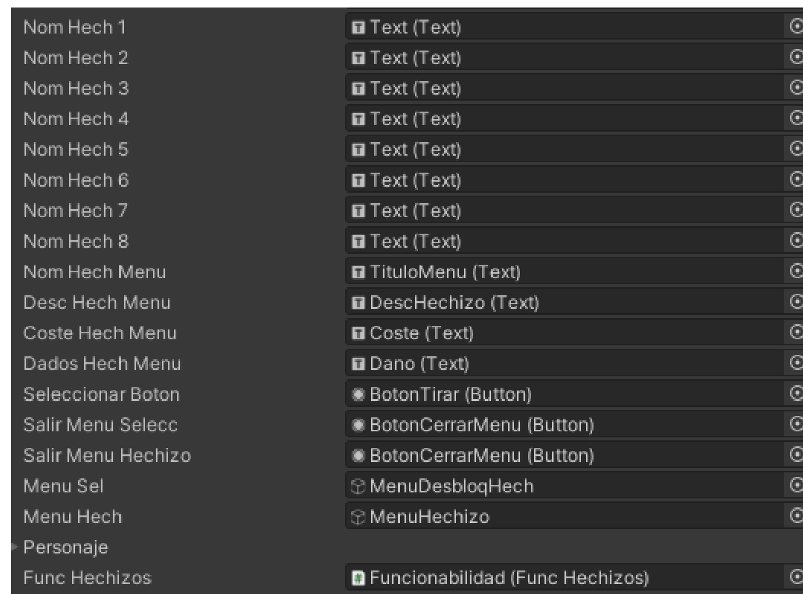

Figura 6.5.7/6B: Captura del script y sus atributos.

Sus funciones son las siguientes:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder manipular los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **Start():** Inicializa y asigna a los botones de desbloqueo y de sus menús funciones para cuando son pulsados.

```
void Start()
{
    boton1.onClick.AddListener(Bypass1);
    boton2.onClick.AddListener(Bypass2);
    boton3.onClick.AddListener(Bypass3);
    boton4.onClick.AddListener(Bypass4);
    boton5.onClick.AddListener(Bypass5);
    boton6.onClick.AddListener(Bypass6);
    boton7.onClick.AddListener(Bypass7);
    boton8.onClick.AddListener(Bypass8);
    hech1.onClick.AddListener(Bypass9);
    hech2.onClick.AddListener(Bypass10);
    hech3.onClick.AddListener(Bypass11);
    hech4.onClick.AddListener(Bypass12);
    hech5.onClick.AddListener(Bypass13);
    hech6.onClick.AddListener(Bypass14);
    hech7.onClick.AddListener(Bypass15);
    hech8.onClick.AddListener(Bypass16);

    seleccionarBoton.onClick.AddListener(SelecNuevoHech);

    salirMenuSelecc.onClick.AddListener(CerrarMenuSelec);
    salirMenuHechizo.onClick.AddListener(CerrarMenuHech);
}
```

- **IniciarMenuSelecc():** Función usada para preparar el menú de las habilidades por aprender. Primero consulta

los hechizos vinculados a la clase perteneciente al personaje y luego comprueba cuáles son los hechizos que el personaje aún no ha aprendido, desactivando los botones de las habilidades ya aprendidas.

```
 public void IniciarMenuSelecc()
{
    nomHech1.text = Personaje._clase.listaHechizosClase[0].nombreHechizo;
    nomHech2.text = Personaje._clase.listaHechizosClase[1].nombreHechizo;
    nomHech3.text = Personaje._clase.listaHechizosClase[2].nombreHechizo;
    nomHech4.text = Personaje._clase.listaHechizosClase[3].nombreHechizo;
    nomHech5.text = Personaje._clase.listaHechizosClase[4].nombreHechizo;
    nomHech6.text = Personaje._clase.listaHechizosClase[5].nombreHechizo;
    nomHech7.text = Personaje._clase.listaHechizosClase[6].nombreHechizo;
    nomHech8.text = Personaje._clase.listaHechizosClase[7].nombreHechizo;

    if(Personaje._clase.listaHechizosClase[0].nombreHechizo == ".")
    {
        hech1GO.SetActive(false);
    }
    if (Personaje._clase.listaHechizosClase[1].nombreHechizo == ".")
    {
        hech2GO.SetActive(false);
    }
    if (Personaje._clase.listaHechizosClase[2].nombreHechizo == ".")
    {
        hech3GO.SetActive(false);
    }
    if (Personaje._clase.listaHechizosClase[3].nombreHechizo == ".")
    {
        hech4GO.SetActive(false);
    }
    if (Personaje._clase.listaHechizosClase[4].nombreHechizo == ".")
    {
        hech5GO.SetActive(false);
    }
    if (Personaje._clase.listaHechizosClase[5].nombreHechizo == ".")
    {
        hech6GO.SetActive(false);
    }
    if (Personaje._clase.listaHechizosClase[6].nombreHechizo == ".")
    {
        hech7GO.SetActive(false);
    }
    if (Personaje._clase.listaHechizosClase[7].nombreHechizo == ".")
    {
        hech8GO.SetActive(false);
    }
}
```

- **Bypass() 1-8:** Grupo de funciones con el objetivo de iniciar el proceso de aprendizaje de una nueva habilidad. Cuando uno de los botones de los huecos se pulsan, este tipo de función se encarga de guardar el botón pulsado para desactivarlo en la función "SelecNuevoHech" y activar el menú de habilidades por aprender mediante la llamada de la función "EmpezarSelecc".

```
void Bypass1()
{
    EmpezarSelecc(boton1GO);
}
```

- **Bypass() 9-16:** Grupo de funciones con el objetivo de activar el menú de la habilidad seleccionada. Cuando uno de los botones del hueco se pulsa, este tipo de función llama a la función "AbrirMenuHech" pasándole por parámetro el botón pulsado y el número en la lista de habilidades de la clase correspondiente a la habilidad seleccionada.

```
void Bypass9()
{
    AbrirMenuHech(hech1GO,0);
}
```

- **SelecNuevoHech():** Función usada para asignar una habilidad al personaje, guardando sus datos en el listado de habilidades aprendidas del personaje llamado "listaHechizos" y marca la habilidad en el listado de habilidades de la clase como aprendida.

```
void SelecNuevoHech()
{
    int num = Personaje.hechizosGuard;
    string nombreHechizo =
    Personaje._clase.listaHechizosClase[index].nombreHechizo;
    string descHechizo =
    Personaje._clase.listaHechizosClase[index].descHechizo;
    string tipoHechizo =
    Personaje._clase.listaHechizosClase[index].tipoHechizo;
    int costeHechizo =
    Personaje._clase.listaHechizosClase[index].costeHechizo;
    int dadoHechizo =
    Personaje._clase.listaHechizosClase[index].dadoHechizo;
    Debug.Log(num);
    Personaje.listaHechizos[num].nombreHechizo = nombreHechizo;
    Personaje.listaHechizos[num].descHechizo = descHechizo;
    Personaje.listaHechizos[num].tipoHechizo = tipoHechizo;
    Personaje.listaHechizos[num].costeHechizo = costeHechizo;
    Personaje.listaHechizos[num].dadoHechizo = dadoHechizo;
    Personaje._clase.listaHechizosClase[index].nombreHechizo = ".";
    Personaje.hechizosGuard++;
    MenuHech.SetActive(false);
    MenuSel.SetActive(false);
    botonHechAct.SetActive(false);
    botonDesbloqAct.SetActive(false);
    FuncHechizos.ActHechiz(num);
}
```

- **ActHechiz():** Función usada para actualizar el hueco de la habilidad con la información de esta, llamando la función “ActHechiz” del script “FuncHechizos”.

```
public void ActHechiz(int n)
{
    FuncHechizos.ActHechiz(n);
}
```

- **EmpezarSelecc():** Función usada para iniciar el proceso de aprendizaje de una nueva habilidad. Cuando uno de los botones de los huecos se pulsa, este tipo de función se encarga de guardar el botón pulsado para desactivarlo en otra función y activar el menú de habilidades por aprender.

```
void EmpezarSelecc(GameObject botonAct)
{
    botonDesbloqAct = botonAct;
    MenuSel.SetActive(true);
}
```

- **AbrirMenuHech():** Función usada para preparar el menú de una habilidad con su respectiva información y activarlo para hacerlo visible.

```
void AbrirMenuHech(GameObject botonHechizoAct, int indexHech)
{
    index = indexHech;
    botonHechAct = botonHechizoAct;
    nomHechMenu.text =
    Personaje._clase.listaHechizosClase[index].nombreHechizo;
    descHechMenu.text =
    Personaje._clase.listaHechizosClase[index].descHechizo;
    costeHechMenu.text = "Coste: " +
    Personaje._clase.listaHechizosClase[index].costeHechizo + " PA";
    int dado = Personaje._clase.listaHechizosClase[index].dadoHechizo;
    if (dado == 0)
    {
        dadosHechMenu.text = "Dado: Ninguno";
    }
    else
    {
        dadosHechMenu.text = "Dado: d" + dado;
    }
    MenuHech.SetActive(true);
}
```

- **CerrarMenuSelec():** Función usada para cerrar el menú de habilidades por aprender, desactivando el objeto al que pertenece.

```
void CerrarMenuSelec()
{
    MenuSel.SetActive(false);
}
```

- **CerrarMenuSelec():** Función usada para cerrar el menú de una habilidad, desactivando el objeto al que pertenece.

```
void CerrarMenuHech()
{
    MenuHech.SetActive(false);
}
```

Para que el personaje pueda usar una habilidad, el jugador solo tiene que ir al hueco de dicha habilidad, aprendida con anterioridad, y pulsar el botón con el texto “USAR”. Entonces, se muestra un menú con la información de dicha habilidad. En ese menú es donde el usuario decide si ejecutar la habilidad o no, pulsando el botón principal del menú. Ver **Figura 6.5.7/7**.

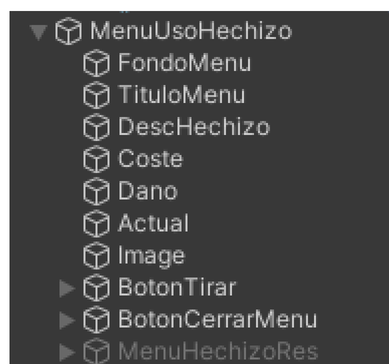

Figura 6.5.7/7: Captura de la jerarquía del menú de una habilidad.

Al usarse una habilidad, se calcula el resultado de la tirada de dados si es necesaria y se muestra un menú con el resultado de esa tirada. Si no es necesario el uso de dados en esa tirada, simplemente se muestra en el menú de resultado una explicación de lo que hace la habilidad. Ver **Figura 6.5.7/8**.

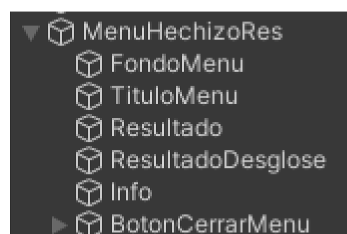

Figura 6.5.7/8: Captura de la jerarquía del menú del resultado del uso de una habilidad.

El comportamiento de el uso de una habilidad, sus botones y menús es controlado por el script “FuncHechizos”. **Figura 6.5.7/9A y 6.5.7/9B**.

Func Hechizos (Script)	
Script	FuncHechizos
Nom Hech 1	TituloPasivas (Text)
Nom Hech 2	TituloPasivas (Text)
Nom Hech 3	TituloPasivas (Text)
Nom Hech 4	TituloPasivas (Text)
Nom Hech 5	TituloPasivas (Text)
Nom Hech 6	TituloPasivas (Text)
Nom Hech 7	TituloPasivas (Text)
Nom Hech 8	TituloPasivas (Text)
Coste Hech 1	TituloPasivas (Text)
Coste Hech 2	TituloPasivas (Text)
Coste Hech 3	TituloPasivas (Text)
Coste Hech 4	TituloPasivas (Text)
Coste Hech 5	TituloPasivas (Text)
Coste Hech 6	TituloPasivas (Text)
Coste Hech 7	TituloPasivas (Text)
Coste Hech 8	TituloPasivas (Text)
Dados Hech 1	TituloPasivas (Text)
Dados Hech 2	TituloPasivas (Text)
Dados Hech 3	TituloPasivas (Text)
Dados Hech 4	TituloPasivas (Text)
Dados Hech 5	TituloPasivas (Text)
Dados Hech 6	TituloPasivas (Text)
Dados Hech 7	TituloPasivas (Text)
Dados Hech 8	TituloPasivas (Text)
Usar Hech 1	BotonUsarHechizo (Button)
Usar Hech 2	BotonUsarHechizo (Button)
Usar Hech 3	BotonUsarHechizo (Button)
Usar Hech 4	BotonUsarHechizo (Button)
Usar Hech 5	BotonUsarHechizo (Button)
Usar Hech 6	BotonUsarHechizo (Button)
Usar Hech 7	BotonUsarHechizo (Button)
Usar Hech 8	BotonUsarHechizo (Button)

Figura 6.5.7/9A: Captura del script y sus atributos.

Menu Hech	MenuUsoHechizo
Nom Hech Menu	TituloMenu (Text)
Desc Hech Menu	DescHechizo (Text)
Coste Hech Menu	Coste (Text)
Dados Hech Menu	Dano (Text)
PA Pers Menu	Actual (Text)
Usar Hech	BotonTirar (Button)
Cerrar Menu Hech	BotonCerrarMenu (Button)
Resultado Hech	MenuHechizoRes
Result Hech	Resultado (Text)
Desglose Result Hech	ResultadoDesglose (Text)
Info Result Hech	Info (Text)
Cerrar Menu Result Hech	BotonCerrarMenu (Button)
PA Pers	Bonus (Text)
Personaje	
Audio Data	AudioDado (Audio Source)

Figura 6.5.7/9B: Captura del script y sus atributos.

Sus funciones son las siguientes:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder manipular los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **Start():** Inicializa y asigna a los botones de desbloqueo y de sus menús funciones para cuando son pulsados. También se encarga de poner las habilidades aprendidas por el personaje en cada hueco del bloque llamando la función "IniciarHechPers".

```
void Start()
{
    usarHech1.onClick.AddListener(Bypass1);
    usarHech2.onClick.AddListener(Bypass2);
    usarHech3.onClick.AddListener(Bypass3);
    usarHech4.onClick.AddListener(Bypass4);
    usarHech5.onClick.AddListener(Bypass5);
    usarHech6.onClick.AddListener(Bypass6);
    usarHech7.onClick.AddListener(Bypass7);
    usarHech8.onClick.AddListener(Bypass8);
    usarHech.onClick.AddListener(UsarHech);
    cerrarMenuHech.onClick.AddListener(CerrarMenuHech);
    cerrarMenuResultHech.onClick.AddListener(CerrarMenuResultHech);
    IniciarHechPers();
}
```

- **IniciarHechPers():** Función encargada de poner las habilidades aprendidas por el personaje en cada hueco del bloque. Funciona habilidad por habilidad, comprobando si es necesario buscar la siguiente o no. Usa la función "MeterNuevoHechizo" pasando por

parámetro toda la información de la habilidad correspondiente al hueco.

```
void IniciarHechPers()
{
    int numHechPers = Personaje.hechizosGuard;
    if (numHechPers >= 1)
    {
        MeterNuevoHechizo(nomHech1,
        Personaje.listaHechizos[0].nombreHechizo, costeHech1,
        Personaje.listaHechizos[0].costeHechizo, dadosHech1,
        Personaje.listaHechizos[0].dadoHechizo);
        if (numHechPers >= 2)
        {
            MeterNuevoHechizo(nomHech2,
            Personaje.listaHechizos[1].nombreHechizo, costeHech2,
            Personaje.listaHechizos[1].costeHechizo, dadosHech2,
            Personaje.listaHechizos[1].dadoHechizo);
            if (numHechPers >= 3)
            {
                MeterNuevoHechizo(nomHech3,
                Personaje.listaHechizos[2].nombreHechizo, costeHech3,
                Personaje.listaHechizos[2].costeHechizo, dadosHech3,
                Personaje.listaHechizos[2].dadoHechizo);
                if (numHechPers >= 4)
                {
                    MeterNuevoHechizo(nomHech4,
                    Personaje.listaHechizos[3].nombreHechizo, costeHech4,
                    Personaje.listaHechizos[3].costeHechizo, dadosHech4,
                    Personaje.listaHechizos[3].dadoHechizo);
                    if (numHechPers >= 5)
                    {
                        MeterNuevoHechizo(nomHech5,
                        Personaje.listaHechizos[4].nombreHechizo,
                        costeHech5, Personaje.listaHechizos[4].costeHechizo,
                        dadosHech5, Personaje.listaHechizos[4].dadoHechizo);
                        if (numHechPers >= 6)
                        {
                            MeterNuevoHechizo(nomHech6,
                            Personaje.listaHechizos[5].nombreHechizo,
                            costeHech6,
                            Personaje.listaHechizos[5].costeHechizo,
                            dadosHech6,
                            Personaje.listaHechizos[5].dadoHechizo);
                            if (numHechPers >= 7)
                            {
                                MeterNuevoHechizo(nomHech7,
                                Personaje.listaHechizos[6].nombreHechizo,
                                costeHech7,
                                Personaje.listaHechizos[6].costeHechizo,
```

```
                                    dadosHech7,
                                    Personaje.listaHechizos[6].dadoHechizo);
                                    if (numHechPers >= 8)
                                    {
                                        MeterNuevoHechizo(nomHech8,
Personaje.listaHechizos[7].nombreHechizo, costeHech8,
Personaje.listaHechizos[7].costeHechizo, dadosHech8,
Personaje.listaHechizos[7].dadoHechizo);
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
```

- **Bypass() 1-8:** Grupo de funciones usadas para abrir el menú de una habilidad en específico. Llama la función "AbrirMenuHechizo" pasando como parámetros la información de la habilidad.

```
    void Bypass1()
    {
        hechSelec = 0;
        AbrirMenuHechizo(Personaje.listaHechizos[hechSelec].nombreHechizo,
                         Personaje.listaHechizos[hechSelec].descHechizo,
                         Personaje.listaHechizos[hechSelec].costeHechizo,
                         Personaje.listaHechizos[hechSelec].dadoHechizo);
    }
```

- **AbrirMenuHechizo():** Función usada para abrir el menú de una habilidad. Primero prepara el menú con la información de dicha habilidad y luego activa el objeto del menú para hacerlo visible.

```
void AbrirMenuHechizo(string nomHechizo, string descHechizo, int
                      costeHechizo, int dadoHechizo)
{
    nomHechMenu.text = nomHechizo;
    descHechMenu.text = descHechizo;
    costeHechMenu.text = "Coste: " + costeHechizo + " PA";
    if (dadoHechizo == 0)
    {
        dadosHechMenu.text = "-";
    }
    else
    {
        dadosHechMenu.text = "Dados: d" + dadoHechizo + " X" +
                             Personaje._otros.numDados;
    }
    PAPersMenu.text = "PA Actuales: " + Personaje.Valor("ALMAACT") + "/" +
                      Personaje.Valor("ALMAMAX");
    menuHech.SetActive(true);
}
```

- **UsarHech():** Función usada para decidir qué función usar para interpretar el uso de una habilidad. Comprueba el tipo de la habilidad y en función a eso llama a una función o otra. También de encarga de calcular el gasto de PA por el uso de la habilidad y a actualizar los elementos que muestran los PA actuales del personaje. Finalmente muestra el resultado en otro menú.

```
void UsarHech()
{
    if(Personaje.listaHechizos[hechSelec].tipoHechizo == "Magia")
    {
        UsarMagia(Personaje.listaHechizos[hechSelec].dadoHechizo);
    }
    else if (Personaje.listaHechizos[hechSelec].tipoHechizo == "Ataque")
    {
        UsarAtaque();
    }
    else if (Personaje.listaHechizos[hechSelec].tipoHechizo == "Carga")
    {
        UsarCarga(Personaje.listaHechizos[hechSelec].dadoHechizo);
    }
    else if (Personaje.listaHechizos[hechSelec].tipoHechizo == "Oblit")
    {
        UsarObliteracion();
    }
    else if (Personaje.listaHechizos[hechSelec].tipoHechizo == "Bi")
```

```
        {
            UsarBiataque();
        }
        else if (Personaje.listaHechizos[hechSelec].tipoHechizo == "Combo")
        {
            UsarCombo();
        }
        else if (Personaje.listaHechizos[hechSelec].tipoHechizo == "Mimet")
        {
            UsarMimetizar(Personaje.listaHechizos[hechSelec].descHechizo);
        }
        else if (Personaje.listaHechizos[hechSelec].tipoHechizo == "Pasivo")
        {
            UsarPasivo(Personaje.listaHechizos[hechSelec].descHechizo);
        }
        int PAAct = Personaje.Valor("ALMAACT");
        int valorRestar = Personaje.listaHechizos[hechSelec].costeHechizo;
        int resultado = PAAct - valorRestar;
        Personaje.ActualizarValor("ALMAACT", resultado);
        PAPersMenu.text = "PA Actuales: " + Personaje.Valor("ALMAACT") + "/" +
                           Personaje.Valor("ALMAMAX");
        PAPers.text = Personaje.Valor("ALMAACT") + "/" +
                      Personaje.Valor("ALMAMAX");
        resultadoHech.SetActive(true);

    }
```

- **UsarMagia():** Función encargada del uso de una habilidad de tipo "magia". Calcula la tirada y modifica los elementos del menú del resultado de la habilidad con el resultado de la tirada de dados.

```
    void UsarMagia(int dado)
    {
        int min = 1;
        int max = dado + 1;
        int numDadoTirada;
        int result = 0;
        audioData.Play(0);
        int bonoActual = Personaje.BonoValor("PODER");
        for (int i = 0; i < Personaje._otros.numDados; i++)
        {
            numDadoTirada = Random.Range(min, max);
            result = result + numDadoTirada;
            desgloseResultHech.text += numDadoTirada + " (d" + dado + ") " + "+
                                       ";
        }
        result = result + bonoActual;
```

```
        resultHech.text = "Resultado: " + result;
        desgloseResultHech.text += bonoActual + " (Bono PODER)";
    }
```

- **UsarAtaque():** Función encargada del uso de una habilidad de tipo "ataque". Calcula la tirada y modifica los elementos del menú del resultado de la habilidad con el resultado de la tirada de dados.

```
    void UsarAtaque()
    {
        int min = 1;
        int max = Personaje.ArmaEquipada.dadoArma + 1;
        int numDadoTirada;
        int result = 0;
        audioData.Play(0);
        int bonoActual = Personaje.BonoValor(Personaje.ArmaEquipada.tipoArma);
        for (int i = 0; i < Personaje._otros.numDados; i++)
        {
            numDadoTirada = Random.Range(min, max);
            result = result + numDadoTirada;
            desgloseResultHech.text += numDadoTirada + " (" +
                                       Personaje.ArmaEquipada.dadoArma + ") " +
                                       "+ ";
        }
        result = result + bonoActual;
        if (Personaje.cargado)
        {
            result = result + Personaje.valorCargado;
            desgloseResultHech.text += bonoActual + " (Bono " +
                                       Personaje.ArmaEquipada.tipoArma + ") + "
                                       + Personaje.valorCargado + " (Ataque
                                       cargado)";
            Personaje.valorCargado = 0;
            Personaje.cargado = false;
        }
        else
        {
            desgloseResultHech.text += bonoActual + " (Bono " +
                                       Personaje.ArmaEquipada.tipoArma + ") ";
        }
        resultHech.text = "Resultado: " + result;
    }
```

- **UsarCarga():** Función encargada del uso de la habilidad "Carga". Calcula la tirada y modifica los elementos del

menú del resultado de la habilidad con el resultado de la tirada de dados.

```
void UsarCarga(int dado)
    {
        int min = 1;
        int max = dado + 1;
        int numDadoTirada;
        int result = 0;
        audioData.Play(0);
        int bonoActual = Personaje.BonoValor("PODER");
        for (int i = 0; i < Personaje._otros.numDados; i++)
        {
            numDadoTirada = Random.Range(min, max);
            result = result + numDadoTirada;
            desgloseResultHech.text += numDadoTirada + " (d" + dado + ") " + "+
                                       ";
        }

        result = result + bonoActual;
        Personaje.cargado = true;
        Personaje.valorCargado = result;
        resultHech.text = "Resultado: " + result;
        desgloseResultHech.text += bonoActual + " (Bono PODER)";
    }
```

- **UsarObliteracion():** Función encargada del uso de la habilidad "Obliteración". Calcula la tirada y modifica los elementos del menú del resultado de la habilidad con el resultado de la tirada de dados.

```
    void UsarObliteracion()
    {
        int min = 1;
        int max = Personaje.ArmaEquipada.dadoArma + 1;
        int numDadoTirada;
        int result = 0;
        audioData.Play(0);
        int bonoActual = Personaje.BonoValor(Personaje.ArmaEquipada.tipoArma);
        for (int i = 0; i < Personaje._otros.numDados + 4; i++)
        {
            numDadoTirada = Random.Range(min, max);
            result = result + numDadoTirada;
            desgloseResultHech.text += numDadoTirada + " (" +
                                       Personaje.ArmaEquipada.dadoArma + ") " +
                                       "+ ";
        }
```

```
        result = result + bonoActual;
        if (Personaje.cargado)
        {
            result = result + Personaje.valorCargado;
            desgloseResultHech.text += bonoActual + " (Bono " +
                                       Personaje.ArmaEquipada.tipoArma + ") + "
                                       + Personaje.valorCargado + " (Ataque
                                       cargado)";
            Personaje.valorCargado = 0;
            Personaje.cargado = false;
        }
        else
        {
            desgloseResultHech.text += bonoActual + " (Bono " +
                                       Personaje.ArmaEquipada.tipoArma + ") ";
        }
        resultHech.text = "Resultado: " + result;
    }
```

- **UsarBiataque():** Función encargada del uso de la habilidad "Biataque". Calcula la tirada y modifica los elementos del menú del resultado de la habilidad con el resultado de la tirada de dados.

```
void UsarBiataque()
    {
        int min = 1;
        int max = Personaje.ArmaEquipada.dadoArma + 1;
        int numDadoTirada;
        int result = 0;
        audioData.Play(0);
        int bonoActual = Personaje.BonoValor(Personaje.ArmaEquipada.tipoArma);
        for (int a = 0; a < 2; a++)
        {
            for (int i = 0; i < Personaje._otros.numDados; i++)
            {
                numDadoTirada = Random.Range(min, max);
                result = result + numDadoTirada;
                desgloseResultHech.text += numDadoTirada + " (d" +
                                           Personaje.ArmaEquipada.dadoArma + ")
                                           " + "+ ";
            }
            result = result + bonoActual;

            if (Personaje.cargado)
            {
                result = result + Personaje.valorCargado;
                desgloseResultHech.text += bonoActual + " (Bono " +
```

```
                                                    Personaje.ArmaEquipada.tipoArma + ")
                                                    + " + Personaje.valorCargado + "
                                                    (Ataque cargado) ";
                Personaje.valorCargado = 0;
                Personaje.cargado = false;
            }
            else
            {
                desgloseResultHech.text += bonoActual + " (Bono " +
                                                    Personaje.ArmaEquipada.tipoArma + ")
                                                    ";
            }
            if (a < 1)
            {
                desgloseResultHech.text += "| ";
            }
        }
        resultHech.text = "Resultado: " + result;
    }
```

- **UsarMimetizar():** Función encargada del uso de la habilidad "Mimetizar". Calcula la tirada y modifica los elementos del menú del resultado de la habilidad con el resultado de la tirada de dados.

```
    void UsarMimetizar(string descripcion)
    {
        infoResultHech.text = descripcion;
        Personaje.oculto = true;
    }
```

- **UsarPasivo():** Función encargada del uso de una habilidad de tipo "pasiva". Calcula la tirada y modifica los elementos del menú del resultado de la habilidad con el resultado de la tirada de dados.

```
    void UsarPasivo(string descripcion)
    {
        infoResultHech.text = descripcion;
    }
```

- **UsarCombo():** Función encargada del uso de la habilidad "Combo". Calcula la tirada y modifica los elementos del

menú del resultado de la habilidad con el resultado de la tirada de dados.

```
void UsarCombo()
    {
        int minAtk = 1;
        int maxAtk = Personaje.ArmaEquipada.dadoArma + 1;
        int minBonus = 1;
        int maxBonus = 7;
        int minTimes = Personaje._otros.numDados;
        int maxTimes = 9;
        int numDadoAtaque;
        int numVecesAtaque;
        int numDadoBono;
        int result = 0;
        audioData.Play(0);
        int bonoPoder = Personaje.BonoValor("PODER");
        int bonoActual = Personaje.BonoValor(Personaje.ArmaEquipada.tipoArma);
        numVecesAtaque = Random.Range(minTimes, maxTimes);
        for (int a = 0; a < numVecesAtaque; a++)
        {
            for (int i = 0; i < Personaje._otros.numDados; i++)
            {
                numDadoAtaque = Random.Range(minAtk, maxAtk);
                numDadoBono = Random.Range(minBonus, maxBonus);
                result = result + numDadoAtaque + numDadoBono;
                desgloseResultHech.text += numDadoAtaque + " (d" +
Personaje.ArmaEquipada.dadoArma + ")" + " + " + numDadoBono + " (Daño Adicional)
";
            }
            result = result + bonoActual;

            if (Personaje.cargado)
            {
                result = result + Personaje.valorCargado;
                desgloseResultHech.text += bonoActual + " (Bono " +
Personaje.ArmaEquipada.tipoArma + ") + " + Personaje.valorCargado + " (Ataque
cargado) ";
                Personaje.valorCargado = 0;
                Personaje.cargado = false;
            }
            else
            {
                desgloseResultHech.text += bonoActual + " (Bono " +
Personaje.ArmaEquipada.tipoArma + ") ";
            }
            if (a < numVecesAtaque - 1)
            {
```

```
                desgloseResultHech.text += "| ";
            }
        }
        resultHech.text = "Resultado: " + result;
    }
```

- **ActHechiz():** Función encargada de actualizar la información mostrada en el hueco estipulado por el parámetro de la función "num". Usa la función "MeterNuevoHechizo".

```
public void ActHechiz(int num)
    {
        if(num == 0)
        {
            int dado = Personaje.listaHechizos[0].dadoHechizo;
            MeterNuevoHechizo(nomHech1,
                              Personaje.listaHechizos[0].nombreHechizo,
                              costeHech1,
                              Personaje.listaHechizos[0].costeHechizo,
                              dadosHech1, dado);
        }
                                        . . .
```

Esta comprobación la hace para saber en qué hueco debe actualizar la información, y el patrón se repite para cada hueco.

- **ActHechiz():** Función encargada de actualizar la información mostrada en un hueco.

```
    void MeterNuevoHechizo(Text nombrePoner, string nombre, Text costePoner, int
                           coste, Text dadosPoner, int dados)
    {
        nombrePoner.text = nombre;
        costePoner.text = coste + " PA";

        if (dados == 0)
        {
            dadosPoner.text = "-";
        }
        else
        {
            dadosPoner.text = "d" + dados + " X" + Personaje._otros.numDados;
        }
```

```
}
```

- **CerrarMenuHech():** Función usada para cerrar el menú de una habilidad, desactivando el objeto al que pertenece.

```
void CerrarMenuHech()
{
    menuHech.SetActive(false);
}
```

- **CerrarMenuResultHech():** Función usada para cerrar el menú del resultado del uso de una habilidad, desactivando el objeto al que pertenece.

```
void CerrarMenuResultHech()
{
    resultHech.text = " ";
    desgloseResultHech.text = " ";
    infoResultHech.text = " ";
    resultadoHech.SetActive(false);
}
```

6.5.8. Equipo

Es el bloque donde los jugadores pueden consultar y modificar el inventario y dinero que tiene el personaje. Ver **Figura 6.5.8/1**.

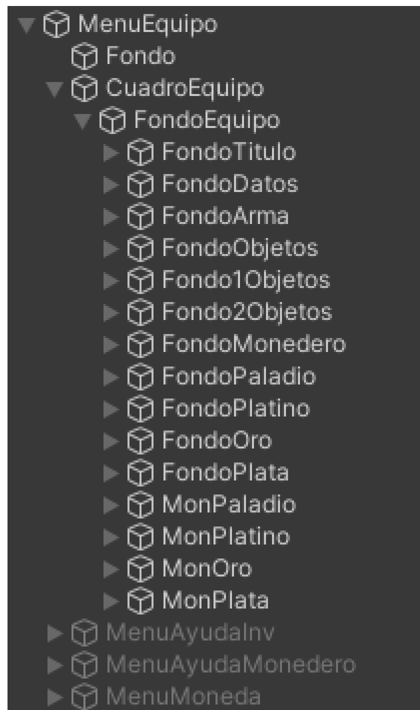

Figura 6.5.8/1: Captura de la jerarquía del bloque del equipo del personaje.

En este bloque se puede encontrar la información del arma, que técnicamente forma parte del equipamiento del personaje, el listado de los objetos que tiene el personaje, y el monedero. Ver **Figura 6.5.8/2**.

Figura 6.5.8/2: Captura de la jerarquía de la información del arma equipada.

Tal y como lo indica el sistema, el personaje puede llevar hasta 8 objetos. Estos se muestran en 8 bloques de texto diferentes, y el usuario puede escribir directamente cada objeto en cada bloque. Ver **Figura 6.5.8/3**.

Figura 6.5.8/3: Captura de la jerarquía del listado de objetos del personaje.

El usuario también puede consultar una descripción de lo que representan y cómo usar los bloques de texto pulsando el icono de pregunta encontrado en la parte superior derecha del inventario. Al hacerlo, se muestra un menú con esa información. Ver **Figura 6.5.8/4**.

Figura 6.5.8/4: Captura de la jerarquía del menú de ayuda del inventario.

Para que se guarden los objetos cuando se escriben y poder mostrar el menú al pulsar el icono, se usa el script "Inventario". Ver **Figura 6.5.8/5**.

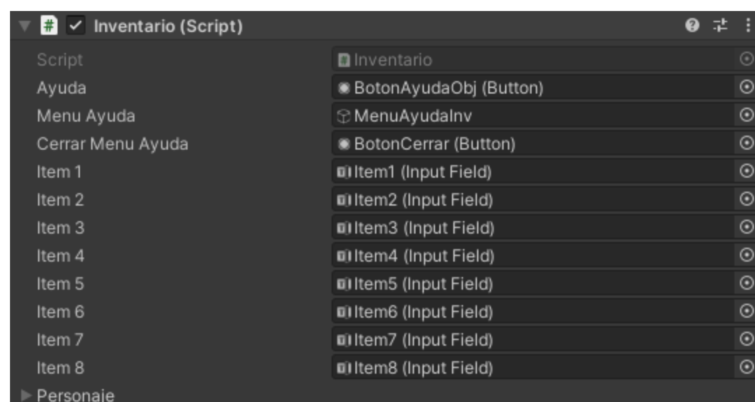

Figura 6.5.8/5: Captura del script y sus atributos.

Sus funciones son:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder acceder a los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **Start():** Función usada para inicializar y asignar a los botones del icono y menú las funciones para cuando se pulsan. También se encarga de cargar los objetos del personaje en sus respectivos bloques mediante la función "CargarObjPers". Además, inicializa los InputFields para detectar cuando el usuario los pulsa para saber cuándo guardar los objetos.

```
void Start()
{
    Ayuda.onClick.AddListener(AbrirMenuAyu);
    CerrarMenuAyuda.onClick.AddListener(CerrarMenuAyu);
    CargarObjPers();
    item1.onValueChanged.AddListener(delegate { OnValueChangedItem1(); });
    item2.onValueChanged.AddListener(delegate { OnValueChangedItem2(); });
    item3.onValueChanged.AddListener(delegate { OnValueChangedItem3(); });
    item4.onValueChanged.AddListener(delegate { OnValueChangedItem4(); });
    item5.onValueChanged.AddListener(delegate { OnValueChangedItem5(); });
    item6.onValueChanged.AddListener(delegate { OnValueChangedItem6(); });
    item7.onValueChanged.AddListener(delegate { OnValueChangedItem7(); });
    item8.onValueChanged.AddListener(delegate { OnValueChangedItem8(); });
}
```

- **CargarObjPers():** Función usada para cargar los objetos del personaje y asignarlos en sus respectivos bloques.

```
void CargarObjPers()
{
    item1.text = Personaje.listaInventario[0];
    item2.text = Personaje.listaInventario[1];
    item3.text = Personaje.listaInventario[2];
    item4.text = Personaje.listaInventario[3];
    item5.text = Personaje.listaInventario[4];
    item6.text = Personaje.listaInventario[5];
    item7.text = Personaje.listaInventario[6];
    item8.text = Personaje.listaInventario[7];
}
```

- **OnValueChangedItem() 1-8:** Grupo de funciones que se ejecutan al pulsar en su respectivo InputField. Se encarga de guardar en el personaje los objetos escritos en ellos.

```
public void OnValueChangedItem1()
{
    Personaje.listaInventario[0] = item1.text;
}
```

- **AbrirMenuAyu():** Función usada para abrir el menú de ayuda, activando el objeto al que pertenece.

```
void AbrirMenuAyu()
{
    menuAyuda.SetActive(true);
}
```

- **CerrarMenuAyu():** Función usada para cerrar el menú de ayuda, desactivando el objeto al que pertenece.

```
void CerrarMenuAyu()
{
    menuAyuda.SetActive(false);
}
```

Para poder manipular la cantidad de dinero que posee el personaje, el usuario puede acceder al monedero, representado por 4 botones, uno por cada tipo de moneda del sistema de rol

y donde se muestra la cantidad exacta del tipo de moneda correspondiente que posee. Ver **Figura 6.5.8/6**.

Figura 6.5.8/6: Captura de la jerarquía del monedero.

Cuando se pulsa en cualquiera de esos botones, aparece el menú de la moneda correspondiente, con información y un InputField donde insertar el número de monedas que se quieren añadir o restar de ese tipo. Ver **Figura 6.5.8/7**.

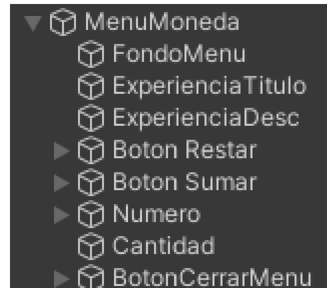

Figura 6.5.8/7: Captura de la jerarquía del menú de una moneda.

Además, el usuario tiene la posibilidad de consultar información sobre el uso del monedero y las monedas pulsando el icono de interrogación encontrado en la esquina superior derecha del monedero. Si lo hace, se abre un menú mostrando esa información. Ver **Figura 6.5.8/8**.

Figura 6.5.8/8: Captura de la jerarquía del menú de ayuda del monedero.

La funcionalidad de todos los elementos del monedero está controlada por el script “MonederoFunc”. Ver **Figura 6.5.8/9**.

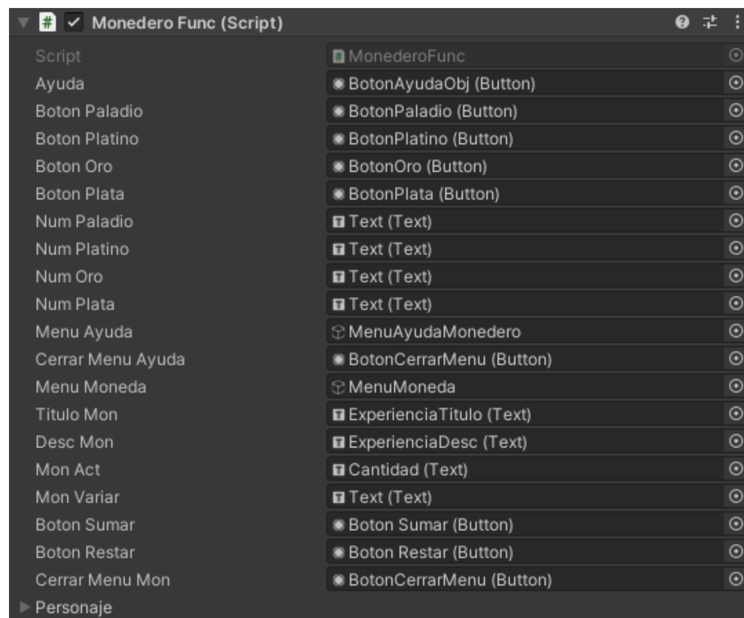

Figura 6.5.8/9: Captura del script y sus atributos.

Sus funciones son las siguientes:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder acceder a los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **Start():** Función usada para inicializar y asignar a los botones del monedero las funciones para cuando se pulsan. También se encarga de cargar el número de monedas de cada tipo y ponerlos en sus respectivos elementos usando la función “IniciarMonedas”.

```
void Start()
{
    IniciarMonedas();
    BotonSumar.onClick.AddListener(SumarMoneda);
    BotonRestar.onClick.AddListener(RestarMoneda);
    CerrarMenuMon.onClick.AddListener(CerrarMenuMonedas);
    Ayuda.onClick.AddListener(AbrirMenuAyu);
    CerrarMenuAyuda.onClick.AddListener(CerrarMenuAyu);
}
```

- **IniciarMonedas():** Función usada para cargar el número de monedas de cada tipo y ponerlos en sus respectivos botones además de inicializarlos y asignarles funciones para cuando se pulsen.

```
void IniciarMonedas()
{
    Paladio = Personaje._monedero.Paladio;
    Platino = Personaje._monedero.Platino;
    Oro = Personaje._monedero.Oro;
    Plata = Personaje._monedero.Plata;
    NumPaladio.text = Paladio.ToString();
    NumPlatino.text = Platino.ToString();
    NumOro.text = Oro.ToString();
    NumPlata.text = Plata.ToString();
    BotonPaladio.onClick.AddListener(MenuPaladio);
    BotonPlatino.onClick.AddListener(MenuPlatino);
    BotonOro.onClick.AddListener(MenuOro);
    BotonPlata.onClick.AddListener(MenuPlata);
}
```

- **Menu*Moneda*():** Conjunto de funciones usadas para preparar el menú de las monedas con la información de la moneda a la que corresponde el botón pulsado. Hecho eso, abre el menú de la moneda.

```
void MenuPaladio()
{
    TituloMon.text = "PALADIO";
    DescMon.text = "La moneda mas valiosa de todas. Su uso se reserva para
                    transacciones de propiedades de altísimo valor. Una
                    moneda de Paladio equivale a 10 de Platino.";
    MonAct.text = "Cantidad Actual: " + Paladio;
    menuMoneda.SetActive(true);
}
```

- **SumarMoneda():** Función usada para aumentar el número de una moneda específicamente. Para saber qué moneda se está modificando, comprueba la moneda que se muestra en el menú. Para la modificación, llama la función "ActMonederoPers".

```
void SumarMoneda()
{
    int valorEntrado;
    int.TryParse(MonVariar.text, out valorEntrado);
    if(TituloMon.text == "PALADIO")
    {
        Paladio = Paladio + valorEntrado;
        MonAct.text = "Cantidad Actual: " + Paladio;
        NumPaladio.text = Paladio.ToString();
    }
    else if (TituloMon.text == "PLATINO")
    {
        Platino = Platino + valorEntrado;
        MonAct.text = "Cantidad Actual: " + Platino;
        NumPlatino.text = Platino.ToString();
    }
    else if (TituloMon.text == "ORO")
    {
        Oro = Oro + valorEntrado;
        MonAct.text = "Cantidad Actual: " + Oro;
        NumOro.text = Oro.ToString();
    }
    else if (TituloMon.text == "PLATA")
    {
        Plata = Plata + valorEntrado;
        MonAct.text = "Cantidad Actual: " + Plata;
        NumPlata.text = Plata.ToString();
    }
    ActMonederoPers();
}
```

- **RestarMoneda():** Función usada para reducir el número de una moneda específicamente. Para saber qué moneda se está modificando, comprueba la moneda que se muestra en el menú. Para la modificación, llama la función "ActMonederoPers".

```
void RestarMoneda()
{
    int valorEntrado;
    int.TryParse(MonVariar.text, out valorEntrado);
    if (TituloMon.text == "PALADIO")
    {
        if( valorEntrado >= Paladio) { Paladio = 0; }
        else { Paladio = Paladio - valorEntrado; }

        MonAct.text = "Cantidad Actual: " + Paladio;
        NumPaladio.text = Paladio.ToString();
    }
    else if (TituloMon.text == "PLATINO")
    {
        if (valorEntrado >= Platino) { Platino = 0; }
        else { Platino = Platino - valorEntrado; }
        MonAct.text = "Cantidad Actual: " + Platino;
        NumPlatino.text = Platino.ToString();
    }
    else if (TituloMon.text == "ORO")
    {
        if (valorEntrado >= Oro) { Oro = 0; }
        else { Oro = Oro - valorEntrado; }
        MonAct.text = "Cantidad Actual: " + Oro;
        NumOro.text = Oro.ToString();
    }
    else if (TituloMon.text == "PLATA")
    {
        if (valorEntrado >= Plata) { Plata = 0; }
        else { Plata = Plata - valorEntrado; }
        MonAct.text = "Cantidad Actual: " + Plata;
        NumPlata.text = Plata.ToString();
    }
    ActMonederoPers();
}
```

- **CerrarMenuMonedas():** Función usada para cerrar el menú de una moneda, desactivando el objeto al que pertenece.

```
void CerrarMenuMonedas()
{
    menuMoneda.SetActive(false);
}
```

- **AbrirMenuAyu():** Función usada para abrir el menú de ayuda, activando el objeto al que pertenece.

```
void AbrirMenuAyu()
{
    menuAyuda.SetActive(true);
}
```

- **CerrarMenuAyu():** Función usada para cerrar el menú de ayuda, desactivando el objeto al que pertenece.

```
void CerrarMenuAyu()
{
    menuAyuda.SetActive(false);
}
```

- **ActMonederoPers():** Función usada para actualizar los valores de las monedas guardadas en el personaje.

```
void ActMonederoPers()
{
    Personaje._monedero.Paladio = Paladio;
    Personaje._monedero.Platino = Platino;
    Personaje._monedero.Oro = Oro;
    Personaje._monedero.Plata = Plata;
}
```

6.5.9. Características

Es el bloque donde los jugadores pueden consultar y modificar las características del personaje, atributos usados para guardar descripciones del comportamiento del personaje para que los jugadores puedan recordar cómo interpretarlos. Ver **Figura 6.5.9/1**.

Figura 6.5.9/1: Captura de la jerarquía del bloque de características.

Este bloque está formado por diferentes InputFields donde se guarda cada característica del personaje. El usuario solo debe pulsar en uno de ellos y escribir para modificar una característica.

Para poder guardar las características cada vez que se modifican, se hace uso del script "Caracteristicas". Ver **Figura 6.5.9/2**.

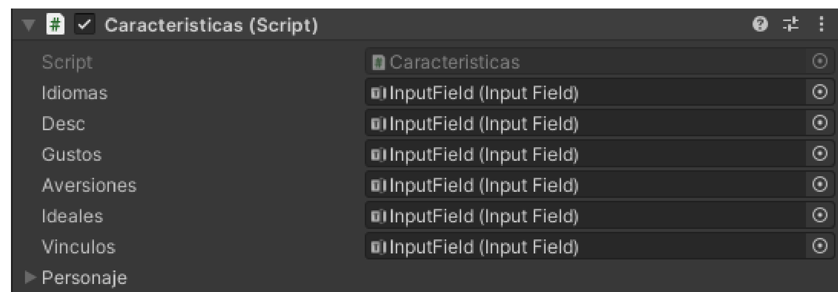

Figura 6.5.9/2: Captura del script y sus atributos.

Sus funciones son:

- **CargarData() y GuardarData():** Funciones de la interfaz de IDataPersistance para poder acceder a los datos del personaje.

```
public void CargarData(Personaje pers)
{
    this.Personaje = pers;
}

public void GuardarData(Personaje pers)
{
    pers = this.Personaje;
}
```

- **Start():** Función usada para cargar las características del personaje en sus respectivos bloques. Además, inicializa los InputFields para detectar cuando el usuario los pulsa, para saber cuándo guardar los objetos.

```
void Start()
{
    idiomas.text = Personaje.idiomas;
    desc.text = Personaje.desc;
    gustos.text = Personaje.gustos;
    aversiones.text = Personaje.aversiones;
    ideales.text = Personaje.ideales;
    vinculos.text = Personaje.vinculos;

    idiomas.onValueChanged.AddListener(delegate { OnValueChangedIdiomas();
                                   });
    desc.onValueChanged.AddListener(delegate { OnValueChangedDesc(); });
    gustos.onValueChanged.AddListener(delegate { OnValueChangedGustos(); });
    aversiones.onValueChanged.AddListener(delegate {
                                      OnValueChangedAversiones(); });
    ideales.onValueChanged.AddListener(delegate { OnValueChangedIdeales();
                                  });
    vinculos.onValueChanged.AddListener(delegate { OnValueChangedVinculos();
                                    });
}
```

- **OnValueChanged*Característica*():** Grupo de funciones que se ejecutan al pulsar en su respectivo InputField. Se encarga de guardar en el personaje los textos escritos en

ellos. La parte en rojo del nombre de la función sustituye al nombre de cada característica.

```
public void OnValueChangedIdiomas()
{
    Personaje.idiomas = idiomas.text;
}
```

6.5.10. Manual del sistema

Es el bloque donde los jugadores pueden consultar el manual del sistema entero sin tener que salir de la aplicación. Ver **Figura 6.5.10/1**.

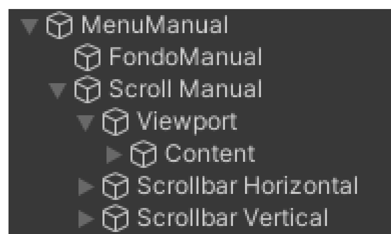

Figura 6.5.10/1: Captura de la jerarquía del bloque del manual del sistema.

Este bloque está formado sólo por un Scroll con un Viewport dentro, y en el objeto "Content" es donde se encuentran las imágenes de todas las páginas del manual juntas. El usuario puede navegar por el manual deslizando el dedo arriba y abajo.

6.6. Pruebas

Durante el desarrollo de la aplicación, se han realizado exhaustivas pruebas cada vez que se implementaban nuevos elementos para comprobar que seguían el comportamiento esperado en todo el sistema. Estas consistían en hacer uso de todos los elementos ya existentes que podrían ser afectados por los cambios nuevos, comprobando en el inspector los cambios en sus atributos durante la ejecución.

También se han ido retocando elementos de diseño a medida que se iban implementando todos los sistemas para lograr una mayor organización visual y mejorar la UX (*User Experience*). Cambios en el tamaño de los textos, colores o tamaños de varios elementos son algunos de esos tipos de ajustes.

Otras pruebas realizadas consistían en simular diferentes tipos de pantalla de varios dispositivos móviles usando el paquete de funcionalidades de Unity “Device Simulator” para comprobar cómo se mostraban los menús en diferentes modelos de dispositivos y hacer correcciones según los resultados. Ver **Figura 6.6/1**.

Figura 6.6/1: A la izquierda, captura de la simulación del gestor de personaje en un Xiaomi Redmi Note 7. A la derecha, captura de la simulación del gestor de personaje en un LG G4.

Para comprender todo lo que pasaba durante la ejecución de varios scripts, se ha hecho uso de las funciones de Debug.Log() y Debug.LogWarning() para ver sus mensajes en la consola del editor y observar los valores que se mostraban.

Para comprobar que la aplicación funcionaba correctamente, se han creado varias *builds* tempranas que se han instalado en un Xiaomi Mi A3 para buscar errores de funcionamiento y solucionarlos. La única forma de solucionar errores que solo ocurren en un dispositivo móvil específico ha sido iterando *builds* nuevas para poder comprobar los cambios implementados.

7. Resultados

7.1. Legislación y normativas vigentes

El sistema de rol Animalia y la aplicación móvil no entran en conflicto con ninguna ley vigente. Como la aplicación móvil no recopila datos de carácter personal ni necesita permisos de usuario, no se le aplica el RGPD (Reglamento General de Protección de Datos), ni la LOPD (Ley Orgánica de Protección de Datos). Como esta aplicación no establece ningún tipo de actividad económica, tampoco se le puede aplicar ninguna ley al respeto como la LSSICE (Ley de Servicios de la Sociedad de la Información y el Comercio Electrónico).

En tema de derechos de autor tampoco puede haber conflicto con la aplicación. Todos los assets usados de terceros han sido obtenidos en páginas web que ofrecen assets gratuitos y de uso libre. El sistema Animalia, aunque esté basado en el sistema de Dragones y Mazmorras, no infringe ninguna ley ya que no toma recursos sinó que comparte un par de mecánicas, por lo tanto, se considera que incorpora la licencia Creative Commons Atributtion 4.0 International, que permite la copia, redistribución y modificación del producto original para cualquier propósito, incluso comercial, siempre y cuando se proporcione crédito a los creadores de la obra original y un enlace de la licencia.

Si se comercializara el proyecto, se debería dar crédito a los creadores de los assets de terceros usados, a los creadores de Dungeons & Dragons, proporcionar la licencia Creative Commons Atributtion 4.0 International, se aplicaría la ley LSSICE y en caso de obtener hasta 100.000€ anuales, sería obligatorio pagar la licencia profesional de Unity, que no limita los ingresos anuales.

7.2. Resultado final

La aplicación ha resultado bastante completa, ofreciendo muchas herramientas que permiten agilizar el proceso no solo de creación de personaje, sino también del uso de las mecánicas que rodean las partidas de rol. El aspecto final de la aplicación puede observarse en el punto 5.3.

Además, en los enlaces a continuación se puede observar el proceso de creación de un personaje usando el creador de personajes de la aplicación y el uso de varias herramientas del gestor de personajes.

- **Creación de un personaje:** https://youtu.be/ATDCNkZiLEA
- **Gestión de un personaje:** https://youtu.be/eSK6DvfnzEM

8. Conclusiones

En el presente punto se analizarán los resultados tras finalizar el proyecto y los cambios que ha sufrido durante su desarrollo respecto a la planificación inicial.

8.1. Valoración final

En general, el resultado de este proyecto es satisfactorio. Se ha conseguido desarrollar no sólo un sistema de rol prácticamente nuevo y funcional por completo, sinó también una aplicación que cumple con la gran mayoría de los requisitos planteados en la fase inicial del proyecto.

Durante el desarrollo se han aprendido muchos aspectos del desarrollo de un proyecto, desde el proceso de diseño hasta la implementación de elementos que interactúan entre sí mismos, se han tenido que aplicar múltiples conocimientos adquiridos durante los cursos previos de este grado y muchos otros aprendidos durante el transcurso de este mismo proyecto.

Por la parte de diseño, los conocimientos obtenidos sobre interfaces de usuario han sido clave para entender cómo estructurar los elementos de los menús de forma que para el usuario sea fácil de entender a primera vista qué representa cada elemento y cómo navegar entre menús. Se podría considerar personalmente como la parte más satisfactoria del desarrollo del proyecto. Iterar varias versiones diferentes de menús, con tamaños de elementos variados y aspectos gráficos diversos ha resultado ser una tarea tanto divertida como laboriosa. A medida que se iban modificando elementos, se ha aprendido cómo de importante son los assets en un proyecto de este estilo, aunque se considera que al final no se ha hecho el uso que se debía para optimizar el diseño de la aplicación lo máximo posible.

En cuanto al diseño gráfico, se han planteado muchos estilos diferentes, pero al final se ha decantado el estilo usado. Ha sido complicado elegir un estilo en concreto, y tener que lograr una cohesión en toda la aplicación ha resultado ser muy complicado, pero al final se valora positivamente el resultado en este aspecto. Le da un punto único y distintivo a la aplicación, y aunque no se considere un

acabado totalmente profesional que podría mejorarse, ya que sigue una línea de diseño completamente opuesta a los cánones actuales del minimalismo, se estima suficientemente sólido como para considerarlo a un buen nivel teniendo en cuenta la experiencia previa en este aspecto y lo logrado al final.

También ha sido muy interesante aprender y entender todas las mecánicas que forman parte de un sistema de rol para poder crear el sistema Animaia, diseñándolo de tal forma que sea algo único, con las reglas impuestas de los números múltiplos de 4 y simplificar sistemas evitando que jugar se vuelva algo aburrido o que caiga en la monotonía demasiado rápido, cosa que se valora haber logrado con creces.

Pasar por el proceso de depuración de una aplicación de estas proporciones ha ayudado a visualizar lo importante que es programar de forma escalable y modificable y siguiendo patrones de diseño de programación. Tener código de esa forma ha ayudado mucho a solucionar errores de forma rápida y efectiva, evitando que un cambio afecte a mucha parte del proyecto. En este aspecto se considera que se podría mejorar mucho, pero al menos se han conocido de primera mano las razones por las cuales seguir esos patrones. A medida que se iban encontrando problemas, se ha tenido que reorganizar partes de código, pero no ha resultado una tarea tan complicada como se esperaba en un principio.

En resumen, se ha trabajado muy duro en este proyecto, aprendiendo a veces por las buenas y a veces por las malas muchos aspectos sobre el desarrollo, sobre todo en Unity. El resultado final, dado la experiencia previa en el momento de iniciar el proyecto y el tiempo disponible, es satisfactorio; aunque como creador, me habría gustado poder dedicarle más tiempo para pulir varios aspectos y ofrecer la mejor versión del proyecto posible.

8.2. Desviaciones de la planificación original

Durante el desarrollo del presente proyecto han aparecido múltiples inconvenientes que han provocado cambios entre la idea original y el resultado final. A medida que iban ocurriendo, han llevado el proyecto por diferentes permutas. las mayores eventualidades ocurridas durante el desarrollo proyecto han sido:

- Trabajar a jornada completa hasta Noviembre, a veces con jornadas de hasta 12 horas. Esto impidió durante muchos días poder trabajar en el proyecto.
- Hacer prácticas a media jornada desde Noviembre hasta la entrega del proyecto. La gran mayoría del proyecto se ha trabajado teniendo menos horas disponibles de las que se podrían haber tenido.
- A mediados de Abril, el PC donde se desarrollaba la aplicación tuvo un fallo crítico y su disco duro SSD, donde se almacenaba el sistema operativo, dejó de funcionar, requiriendo el reemplazo de la pieza y perdiendo todos los datos guardados en este. Por fortuna, solo se perdieron assets no usados, y versiones anteriores, aunque no se pudo utilizar el PC hasta la entrega y montaje del reemplazo de la pieza.

Estos inconvenientes provocaron que varios elementos diseñados para el sistema y la aplicación tuvieran que ser recortados o eliminados para seguir el ritmo de desarrollo acorde con el tiempo disponible. Los cambios principales han sido:

- No desarrollar un sistema para controlar las campañas. Originalmente la aplicación iba a incluir un gestor de campañas para el GM, donde podría administrar elecciones de la historia, localizaciones o el bestiario.
- No diseñar un bestiario. No hubo tiempo para diseñar enemigos especiales para el sistema de rol. Se sustituyeron por personajes iguales a los que controlan los jugadores, pero hostiles a ellos.
- Simplificar el sistema de clases. Originalmente, la clase de un personaje iba a asociarse mediante la elección de 2 valores. Estos podían ser los mismos. La combinación de valores elegida determinaba la clase del personaje. Eso provocó que existieran hasta 56 clases diferentes. Por obvias razones, se tuvo que recortar, dejando solo las clases determinadas al elegir el mismo valor. Aún así, ya existían los nombres y sus descripciones.
- Cambiar el sistema de progresión. Debido al cambio anterior, al subir de nivel se tuvo que cambiar el proceso de elección de los valores a aumentar.
- Eliminar habilidades pasivas y hechizos. Originalmente se había pensado un sistema de habilidades pasivas otorgadas por la

raza y la clase del personaje. Además, las habilidades se llamaban “hechizos”.

- No desarrollar un sistema de elementalidades. Se habían diseñado magias elementales de diversos tipos, pero se tuvieron que descartar por añadir demasiada complejidad al sistema de habilidades y al combate.
- Eliminar los estados alterados. En un principio, las magias elementales podían provocar efectos alterados a los objetivos. Con la eliminación de los elementos, esta mecánica también desapareció.
- No implementar el sistema de tiradas de dados realista. Se quería crear un sistema donde dependiendo de los dados usados en una tirada, aparecieran en pantalla esos dados en forma tridimensional que rebotaran hasta revelar el resultado de la tirada, similar a como se hace en la aplicación de D&D Beyond. La complejidad de desarrollar eso junto a la falta de tiempo imposibilitaron su implementación.
- No diseñar una campaña inicial. Por falta de tiempo, tampoco se creó una campaña con la que los jugadores pudieran empezar a jugar sin tener que idear la historia ellos.

9. Trabajo a futuro

Aunque este proyecto haya culminado en la creación de un sistema de rol completamente funcional y una aplicación que implementa la mayoría de las mecánicas de ese sistema, aún hay muchos otros elementos que se pueden desarrollar para ofrecer un producto mucho más completo:

- **Campañas preestablecidas:** Se podrían desarrollar manuales de campañas con una guía de enemigos, lugares, personajes e historias únicas para que los jugadores puedan jugar sin tener que preocuparse en crear todos esos elementos previamente.
- **Actualizaciones regulares:** Escuchando el feedback de los usuarios, se podrían desarrollar más funcionalidades y mecánicas para la aplicación que pida la comunidad y arreglar bugs no detectados durante la etapa de desarrollo. Además se aprovecharían para pulir el aspecto gráfico de la actualización.
- **Aplicación premium:** Como se ha comentado en el punto 2.4.3, se podría desarrollar una aplicación de pago con funcionalidades extra o ampliadas respecto a la aplicación básica.
- **Versión de iOS:** Se podría desarrollar una versión para los usuarios de iPhone y iPad para así aumentar el posible público del sistema y la aplicación.
- **Nueva edición:** Al igual que otros sistemas de rol como Dragones y Mazmorras, se podría crear una edición nueva del sistema, con más mecánicas como un sistema de fortalezas y debilidades elementales y elementos nuevos como nuevas armas o nuevas razas.
- **Desarrollar el mundo de Animaia:** El mundo del juego es uno de los elementos que menos se ha desarrollado en este proyecto, así que se podría ampliar ese aspecto con un mapamundi, estados con sus regiones, poblaciones y trasfondos de cada uno de ellos.
- **Conectividad entre jugadores y GM:** Siguiendo la sugerencia anterior, se podría desarrollar alguna forma de conectividad entre los jugadores y el GM para que este último reciba y envíe resultados de tiradas para que se gestionen los recursos automáticamente, ya que por ahora se deben notificar los resultados de las tiradas y gestionar esos recursos manualmente.

10. Bibliografía

1. Unity technologies. (s.f.). *Unity Blog*. https://blog.unity.com
2. Unity technologies. (s.f.). *Unity Forum*. https://forum.unity.com
3. Unity technologies. (s.f.). *Unity Answers*. https://answers.unity.com
4. Unity technologies. (s.f.). *Unity Manual*. https://docs.unity3d.com
5. GitHub, Inc. (s.f.). *GitHub*. https://github.com
6. Stack Exchange, Inc. (s.f.). *Stack Overflow*. https://stackoverflow.com
7. Wikimedia Foundation. (s.f.). *Wikipedia, the free encyclopedia*. https://es.wikipedia.org
8. Wizards of the Coast LLC. (s.f.). *Dungeons & Dragons*. https://dnd.wizards.com/es
9. Fandom. (s.f.). *Anima Beyond Fantasy Wiki*. https://anima-beyond-fantasy.fandom.com/es/wiki
10. Trever Mock. (2022, marzo 9). How to make a Save & Load System in Unity | 2022. [Vídeo]. https://youtu.be/aUi9aijvpgs
11. Trever Mock. (2022, abril 18). How to make a Save & Load System work across Multiple Scenes in Unity | 2022 tutorial. [Vídeo]. https://youtu.be/ijVA5Z-Mbh8
12. Trever Mock. (2022, mayo 24). How to Implement Save Slots to Manage Multiple Saved Games in Unity | 2022 tutorial. [Vídeo]. https://youtu.be/Kokt0c8sbNc
13. Trever Mock. (2022, junio 20). Save & Load System in Unity - Bug Fixes, Scriptable Objects, Deleting Data, Backup Files, and More. [Vídeo]. https://youtu.be/yTWPcimAdvY

11. Anexos

Como anexo de este proyecto se han adjuntado el proyecto de la aplicación en Unity y el manual del sistema de rol Animaia. El proyecto en Unity se encuentra dentro del archivo “Animaia V1.0.zip” y para abrirlo se recomienda primero descomprimir el archivo y luego usar el programa “Unity Hub”. El manual se encuentra junto este documento con el nombre “Manual de rol Animaia.pdf”.

12. Manual del usuario e instalación

12.1. Guía de instalación

Para instalar la aplicación, primero es necesario tener un dispositivo Android que tenga la versión 8.0 en adelante.

La aplicación de Animaia se encuentra en el siguiente link:
https://drive.google.com/drive/folders/1-b2B0ZYyQJUvXkJuL-XZTa7_Ah0GGSa6?usp=sharing

Al acceder a ese link usando el navegador de un dispositivo Android, se mostrarán 2 archivos. El primero es la aplicación, con el nombre "AnimaiaApp.apk". El otro archivo es el manual del sistema, llamado "Manual de rol Animaia.pdf". Ver **Figura 12.1/1**.

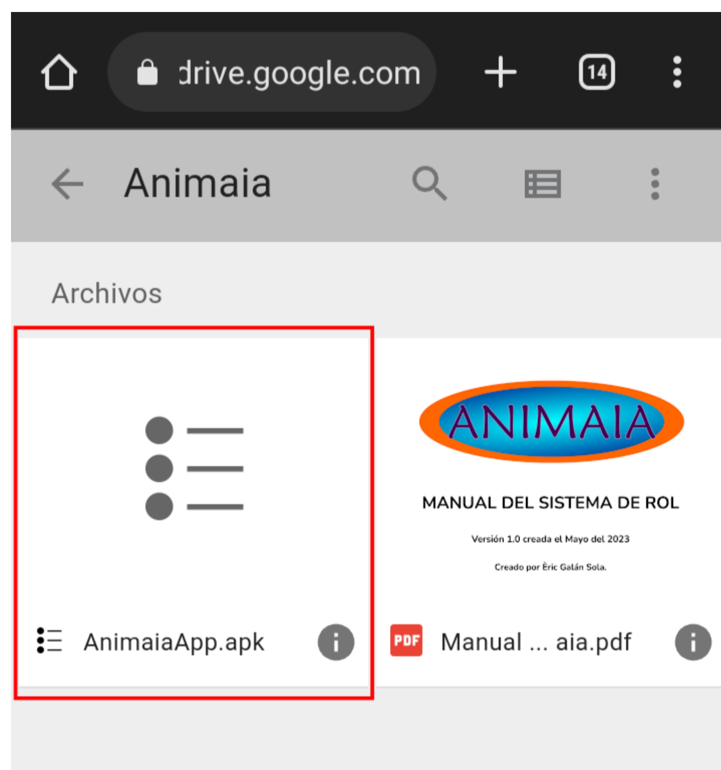

Figura 12.1/1: Captura de la vista de la carpeta de Drive, destacando el archivo de la aplicación.

Al pulsar en el primer archivo, saldrá una vista previa con un botón de descarga. Ver Figura **12.1/2**.

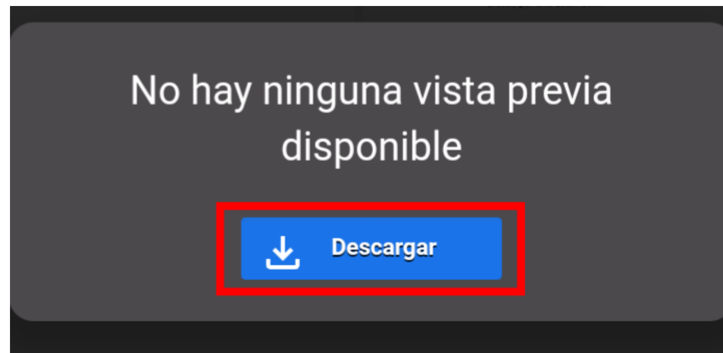

Figura 12.1/2: Captura del mensaje de la vista previa, destacando el botón de descarga.

Al pulsar ese botón, saltará un aviso porque el archivo es demasiado grande para analizarlo como medida preventiva de viruses. En ese momento hay que pulsar el botón en el que pone “Descargar de todos modos”. Ver **Figura 12.1/3**.

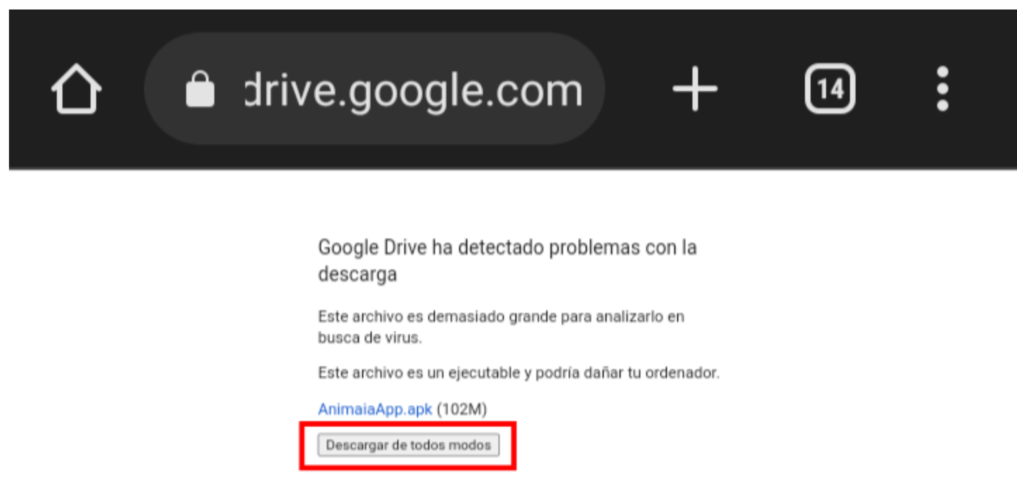

Figura 12.1/3: Captura del primer aviso de descarga peligrosa, destacando al botón que se debe pulsar.

Es probable que vuelva a aparecer otro aviso recordando lo mismo. De nuevo, hay que pulsar el botón en el que pone “Descargar de todos modos”. Ver **Figura 12.1/4**.

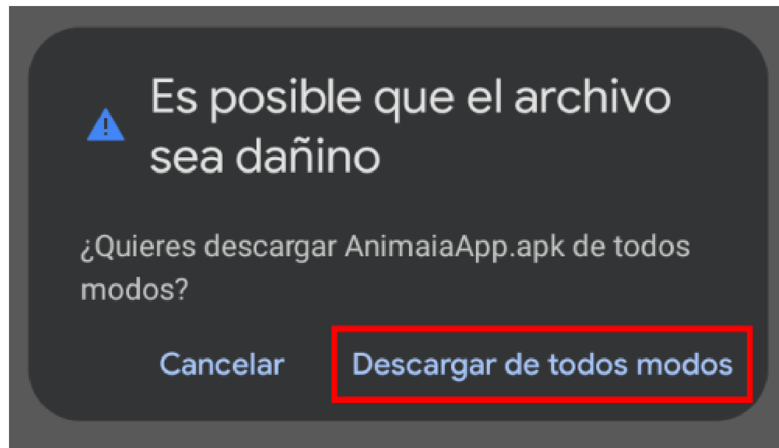

Figura 12.1/4: Captura del segundo aviso de descarga peligrosa, destacando al botón que se debe pulsar.

Con esto, se procederá a la descarga de la aplicación. Cuando termine la descarga, podrá saltar un aviso conforme la descarga ha terminado. Ver **Figura 12.1/5**.

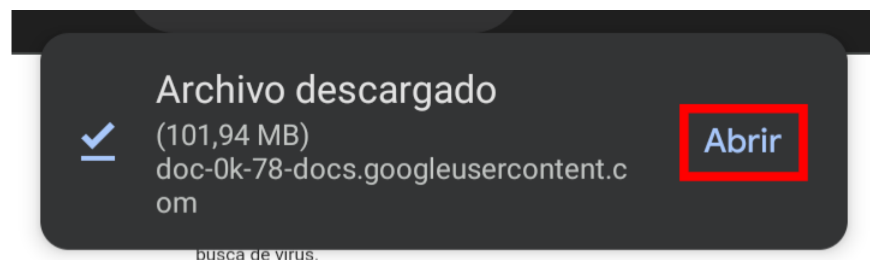

Figura 12.1/5: Captura del aviso de descarga finalizada, destacando dónde se debe pulsar.

Si se pulsa este aviso, aparecerá otra ventana para confirmar la instalación. Aquí hay que confirmar para iniciar la instalación de la aplicación en el dispositivo. Ver **Figura 12.1/6**.

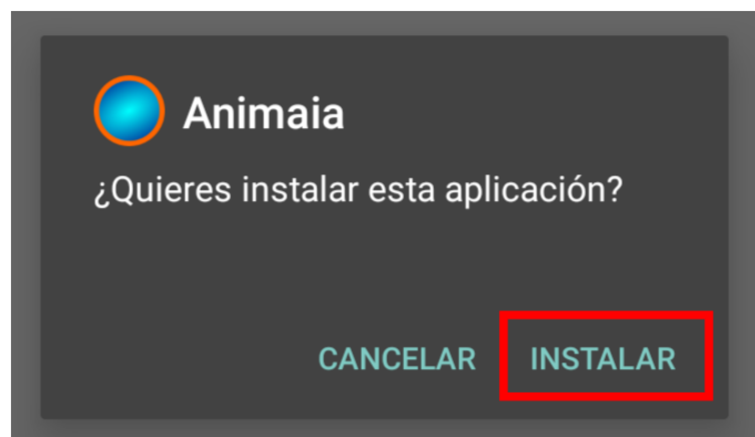

Figura 12.1/6: Captura de la ventana de instalación, destacando dónde se debe pulsar.

También se puede iniciar la instalación accediendo al directorio de descargas del dispositivo y pulsando en el archivo recién descargado.

Cuando termine la instalación, aparecerá una pantalla conforme ha terminado el proceso, dando la opción de cerrar la ventana o abrir la aplicación. Ver **Figura 12.1/7**.

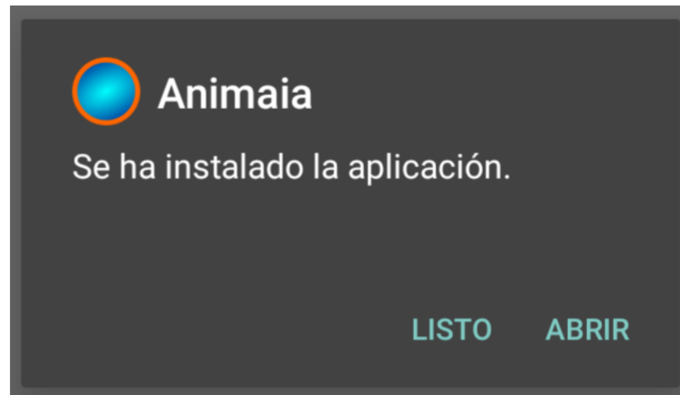

Figura 12.1/7: Captura de la ventana de instalación finalizada.

Aquí finaliza el proceso de descarga e instalación de la aplicación móvil de Animaia. Hecho todo el proceso, la aplicación se podrá encontrar junto al resto de aplicaciones del dispositivo.

12.2. Manual de usuario de la aplicación

Hay varios elementos en los que es necesario hacer un uso específico para que funcionen debidamente.

1. Cuando se quiere aplicar bufos y debufos en un valor del personaje, se debe primero pulsar al valor al que se aplica ese bufo o debufo y poner el número de puntos que aplica la modificación en positivo. Solo es necesario el número el número. Ya con el número puesto, se pulsan los botones “Debuff” o “Buff” dependiendo de lo que se tenga que aplicar. Ver **Figura 12.2/1.**

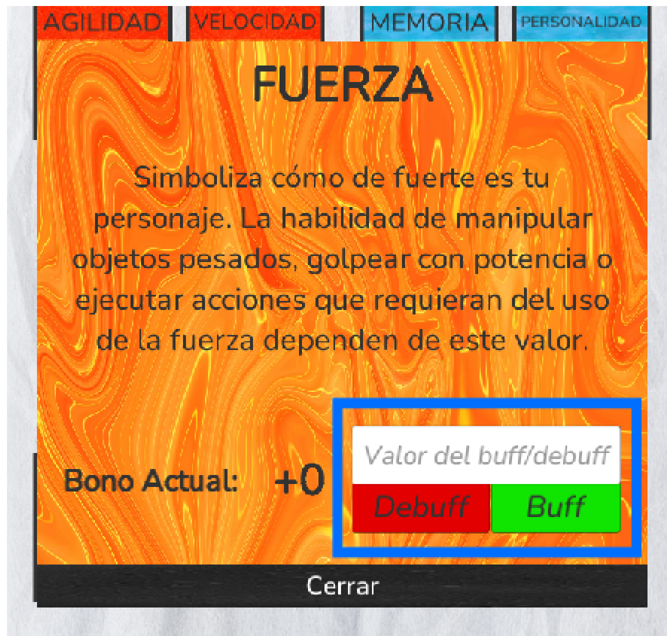

Figura 12.2/1: Captura del menú del valor Fuerza, destacando dónde poner el número.

2. De la misma manera, cuando un personaje recibe daño o una cura, el jugador primero debe pulsar el valor que se modifica, ya sea “Cuerpo” o “Alma”. Dentro de sus menús, en el cuadro de texto, poner el número de puntos de daño o curación en positivo y finalmente, pulsar el botón “Dañar/Usar” o “Curar” según lo que se necesite. Ver **Figura 12.2/2.**

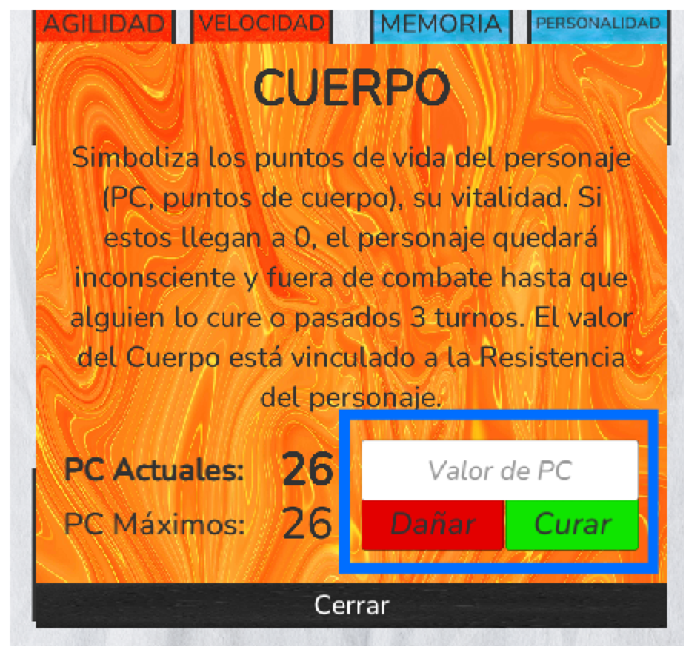

Figura 12.2/2: Captura del menú del valor Cuerpo, destacando dónde poner el número.

3. En combate, cuando se ataca, se debe preguntar al atacado si esquivará o si bloqueará el ataque. Cuando haga una de las dos acciones, se debe pulsar el botón correspondiente a la acción ejecutada por el atacado del mismo menú del resultado de la tirada de dados del ataque. Ver **Figura 12.2/3.**

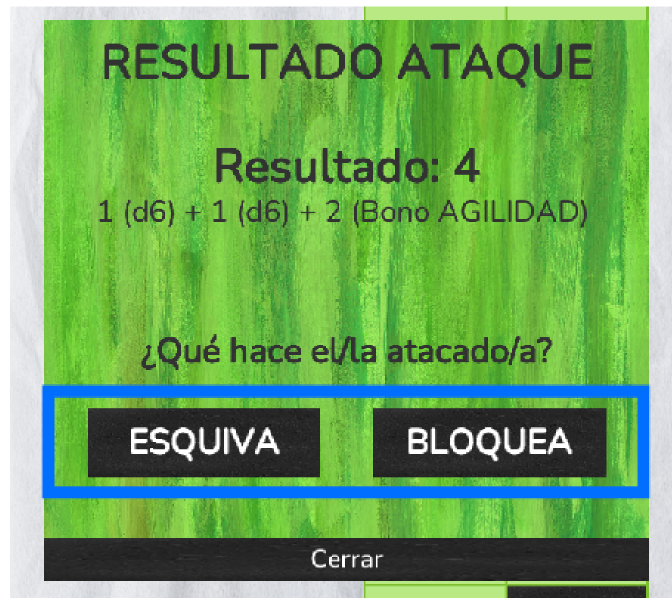

Figura 12.2/3: Captura del menú, destacando los botones que pulsar.

Cuando se pulse uno de los botones, se obtendrá el resultado del atacante y el atacado deberá comunicar el resultado de su tirada. Ese resultado se debe escribir en el cuadro de texto al lado de donde pone “Result. enemigo”. Ese número debe ser positivo y sin signos. Si el resultado es una pifia, se debe escribir “Pifia”. Si el resultado es un milagro, se debe escribir “Milagro”. De lo contrario no se obtendrá un resultado. Ver **Figura 12.2/4.**

Figura 12.2/4: Captura del menú de resultado de esquiva, destacando dónde insertar el resultado del atacado.

4. Si el que está siendo atacado es el jugador, este debe pulsar los botones de esquiva o bloqueo del bloque "Acciones" del gestor de personaje encontrados en la parte inferior de la columna de acciones izquierda. Ver **Figura 12.2/5.**

Figura 12.2/5: Captura del bloque de acciones, destacando los botones que pulsar.

5. Si se usa una habilidad que aplique bufos o debufos, se debe comunicar quién es el objetivo de esa habilidad, los valores que modifica y el número que debe aplicar en el o los valores correspondientes.

6. Si se usa la habilidad “Barrera”, el usuario debe comunicar el resultado y el objetivo debe tener en cuenta ese resultado para saber cuantos puntos extra de vida tiene. Si recibe daño, debe restar ese daño de los puntos de la barrera en lugar de sus PC.

7. Si se usa la habilidad “Imbuir”, el usuario debe tener en cuenta hasta que termine o se actualice el efecto el resultado de la habilidad al calcular el daño que hacen sus ataques con el arma.