# MEMORIA MP

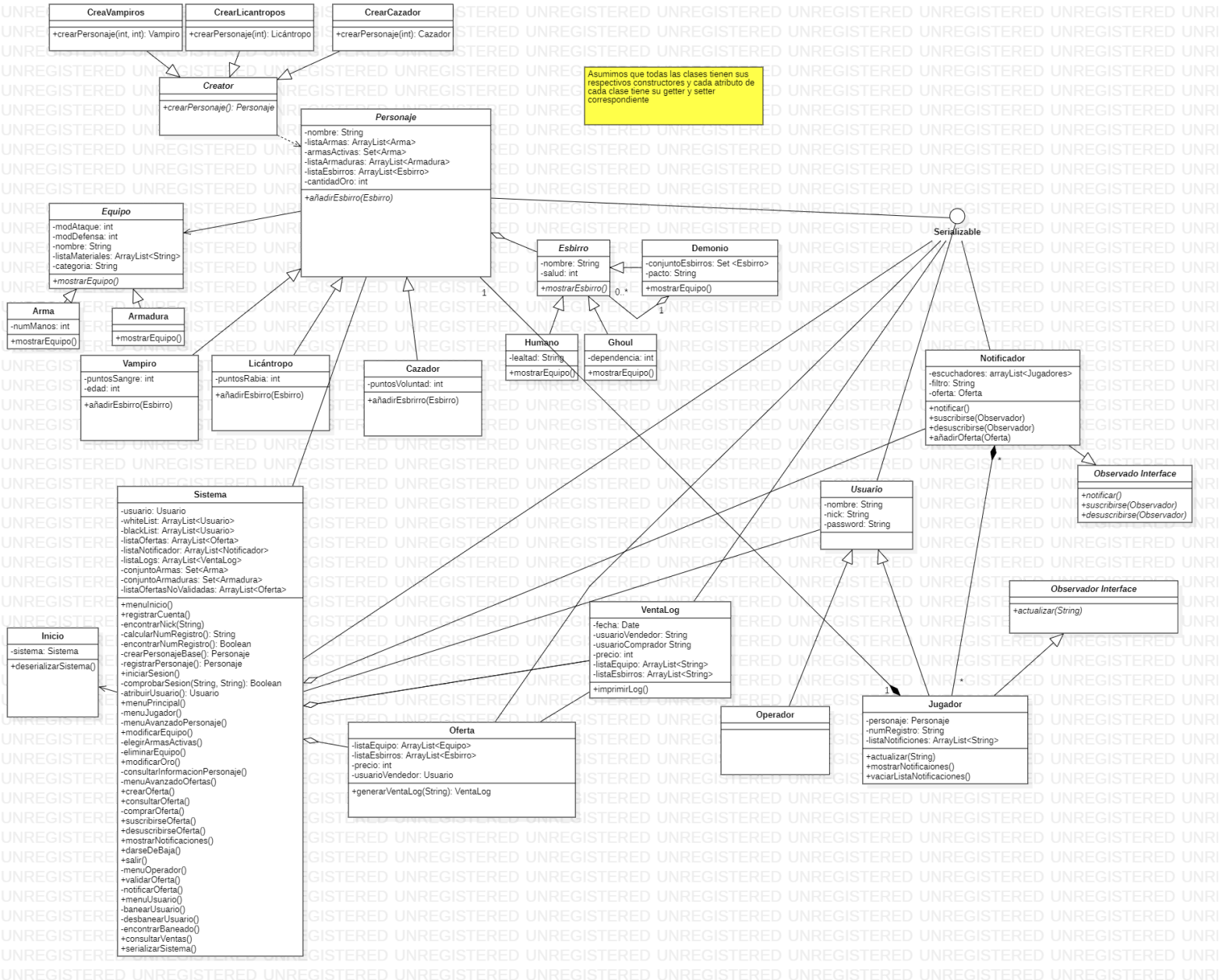
Realizado por:

Frantzes Elzaurdia
Vicente González Pérez
Javier Gaspariño Muñoz
Ángel Covarrubias Roldán

# Índice

## 1. Diseño general y metodología de trabajo.

Para la realización de la práctica empezamos por crear un diagrama UML de clases, para saber de antemano qué clases necesitaremos implementar y qué relación existe entre ellas, así como reconocer qué patrones de diseño deberíamos usar. Como grupo hemos hecho la programación colaborativa mediante el uso de un repositorio de GitHub al que nos conectamos todos mediante la IDE Intellij.




**Mejor abrir el .MDJ o .SVG adjuntado en la carpeta de la práctica para visualizar el diagrama de clases.*

Debemos destacar que nuestra prioridad es crear una aplicación persistente con un diseño que permite añadir contenido con facilidad, por ello creamos diversas clases abstractas e interfaces. Utilizamos el patrón *Factory Method* para la creación de los diferentes personajes y el patrón *Observer* para la gestión de las notificaciones. De esta manera si queremos como desarrolladores añadir algún personaje nuevo, u otro tipo de notificaciones que no sean de ofertas que nos interesen, lo podremos lograr de una manera eficiente. Otro patrón que utilizamos es el *Composite* para la creación de los esbirros.

En cuanto a las clases, Inicio comprueba si existe un fichero para deserializar Sistema que contiene toda la información de la aplicación, si no existe crea un nuevo objeto Sistema. Esta clase se encarga de mostrar los menús y de la navegación y ejecución de todas las operaciones principales del juego. Oferta contiene la información relevante a una oferta creada por un jugador y contiene un método para crear un objeto de tipo VentaLog que almacena además de lo que ya almacena Oferta, el jugador que compró la oferta para almacenar en Sistema el historial de ventas, accesible por un usuario de clase Operador.

Por otro lado, decidimos que todo el equipo existe de forma predeterminada, y que cada jugador puede seleccionar sus armas y armadura de las ya creadas al inicializarse Sistema por primera vez.

En cuanto al funcionamiento de las notificaciones, creamos un objeto Notificador por cada tipo de filtro que los jugadores utilicen para encontrar una oferta que les interesa. Si ya existe un filtro al que te quieres suscribir, te metes a la lista de usuarios que también están suscritos al mismo filtro para evitar crear un número excesivo de objetos Notificador. Cada objeto de este tipo se almacena en una lista de Sistema y son persistentes. Para desuscribirte simplemente te quita de la lista de ese Notificador. Cuando un jugador diferente cree una oferta y un operador la valide, si la oferta contiene un artículo que se ajusta a un filtro al que estás suscrito te enviará una notificación que se guardará en el atributo de Jugador listaNotificaciones, que podrás leer accediendo a “mostrar notificaciones” en el menú principal. Una vez vista la notificación o las notificaciones, se te borrarán. Dentro de la clase Notificador existe el atributo “oferta” para que se pueda comprobar que el usuario vendedor es diferente al que está logueado y no te notifique tu propia oferta.

Otro matiz notable es que para la creación de los demonios, los esbirros hijos que tiene se crean aleatoriamente para evitar tener que introducir la información manualmente y tardar un tiempo excesivo en crear un personaje.

## 2. Tests

Realizamos tests de las partes de la aplicación que consideramos más importantes para su correcto funcionamiento y para cumplir con los requisitos del enunciado de la práctica. En general hicimos tests más largos que envuelven varios tests de una parte del sistema, como por ejemplo un único test exhaustivo del funcionamiento de modificarEquipo(), que implica testear la elección de las armas activas, y el borrado y añadido de equipo al inventario del personaje, así como meter inputs “inesperados”.

Para poder realizar estos tests que comprueban la estructura completa del programa tuvimos que automatizar los inputs del usuario y de las navegaciones por el menú, ya que la mayoría de los métodos son void y no reciben parámetros. Para que esto funcionara, tuvimos que utilizar un único Scanner(System.in) en todo el programa y System.setIn() con un array de bytes con los inputs. De ahí la razón por la cual en todo el código fuente se pasa por parámetro el mismo scanner.

Se utilizan para los tests que requieren inputs, los usuarios genéricos que hagan falta.

La persistencia del programa se ha comprobado manualmente con todos los atributos de la clase Sistema.

### 2.1 nickNoRegistradoTest

Es de caja negra, en concreto de partición equivalente. Las clases de equivalencia evaluadas son:
-Un nick igual a otro usuario creado.
-Un nick distinto a otro usuario creado.
Como es de esperar, en el primer caso no avanza, y te sigue sigue pidiendo introducir un nick correcto. Cuando se introduce un nick válido como en el segundo caso se registra el usuario correctamente.

### 2.2 contraseñaTest

Este test es de caja negra, en concreto de partición de equivalencia. Las clases de equivalencia evaluados son:
-Un valor por debajo del rango (8).
-Un valor por encima del rango (12).

-Un valor entre el rango (8) y (12) de caracteres.
En los dos primeros casos la aplicación manda un mensaje para informar que la contraseña que ha introducido no está entre 8 y 12 caracteres y le pide que introduzca una válida. En el último caso la aplicación sigue funcionando con normalidad.

**2.3 numRegistroAleatorioTest**

Este test es de caja negra, en concreto de partición equivalente. Las clases de equivalencia evaluadas son:
-Un usuario cualquiera.
-Un usuario con distinto número registro a cualquier otro usuario.
Este test comprueba que el número de registro de un usuario al registrarse no coincide con el de cualquier otro usuario creado.

**2.4 crearPersonajeTest**

Este test es de caja blanca ya que su objetivo es comprobar que todos los caminos dentro de crear personaje son válidos y que realizan todos la acción que se corresponde.

**2.5 crearDemonioTest**

Se trata de un test de caja negra que crea dos esbirros de la clase Demonio mediante su constructor y comprueba que sus respectivos conjuntos de esbirros son diferentes entre sí para verificar que se estén generando aleatoriamente.

**2.6 crearOfertaTest**

Este test es de caja blanca ya que su objetivo es comprobar que cuando un jugador crea una oferta, todo los objetos que selecciona se guardan en la estructura de datos listaOfertasNoValidadas, para que después un operador pueda decidir si la oferta se valida o no.

**2.7 menuAvanzadoOfertaTest**

Este test es de caja negra, en concreto de partición de equivalencia. Las clases de equivalencia evaluadas son:
-Un valor por encima del rango (7)
-Un valor por debajo del rango (0)
-Un valor que no es un integer (a)
-Un valor dentro del rango(1)

Como es de esperar, en los tres primeros casos la aplicación espera otra entrada y no avanza, mientras que en el último caso si que avanza al siguiente menú.

**2.8 crearNotificaciónTest**

Este test es de caja blanca ya que su objetivo es comprobar que las estructuras de datos internas guardan bien tanto los notificadores dentro de Sistema como las notificaciones dentro de Personaje. También se encarga de comprobar que las notificaciones se mandan adecuadamente cuando un jugador se suscribe a un tipo de oferta, otro jugador publica una oferta que coincide con la suscripción y un operador valida la oferta adecuadamente.

**2.9 validarOfertaTest**

Es de caja blanca ya que el objetivo de este es comprobar que la lista, (estructura de datos), que contiene las ofertas ya validadas guardan las ofertas que han sido validadas por el operador. Con este test se comprueba de forma indirecta que la creación de las ofertas funciona correctamente ya que, al crear una oferta, esta se guarda en un primer momento en la lista de ofertas a validar y no en la lista de ofertas ya validadas. También se comprueba que si el operador no valida la oferta, esta no aparecerá ni en la lista de ofertas validadas, ni en la lista de ofertas no validadas y se devolverá el equipo ofertado al jugador correspondiente.

**2.10 banearUsuariosTest**

Es un test de caja blanca ya que el objetivo de este es comprobar que la lista que contiene los usuarios baneados guarda el usuario que el operador ha decidido banear. También se comprueba que este usuario salga de la whitelist; es decir, de la lista que contiene los usuarios no baneados.

**2.11 desbanearUsuariosTest**

Es de caja blanca y es bastante similar al anterior test ya que con este se quiere comprobar que un usuario que se encontraba baneado al ser desbaneado por un operador se quita de la blacklist y se introduce en la whitelist junto a los usuarios que no están baneados.

**2.12 darseDeBajaOperadorTest**

Este test es de caja blanca ya que su objetivo es comprobar que, el operador al darse de baja, desaparece whitelist y ya no puede iniciar sesión ya que no existe su cuenta.

**2.13 darseDeBajaJugadorTest**

Consiste en un test de caja blanca en el que se comprueba que, el jugador al darse de baja desaparece de la whitelist y ya no puede acceder a su cuenta iniciando sesión ya que no existirá esta.

**2.14 buscarComprarOfertaTest**

Consiste en un test de caja blanca que comprueba que el usuario puede comprar una oferta y a su vez esta desaparece de la lista de compras, junto a la comprobación de que el log de esta venta se ha guardado en la lista de logs de Sistema. Aparte, se comprueba indirectamente todo lo que se ha comprobado en el test de validarOferta.

**2.15 modificarOroTest**

Este test de caja blanca asigna al personaje 500 de oro, le resta 700 y finalmente le suma 100. Asegura que el oro del personaje nunca es negativo y en el camino demuestra que no se pueden sumar ni restar dentro de modificarOro() un número fuera del rango [0, 1000], tanto por encima como por debajo.

**2.16 modificarEquipoTest**

A lo largo de este test de caja blanca se comprueba que al eliminar un arma de tu inventario general, te lo borra también de tus armas activas. Se intenta eliminar equipo cuando no tienes nada en el inventario y no te permite. Te añade dos armas y te las asigna como armas activas, comprobándose que no puedes tener más de dos. Por lo tanto demuestra el correcto funcionamiento de los métodos modificarEquipo(), añadirEquipo(), elegirArmasActivas() y eliminarEquipo().