Gladiathor the Fool

Materia: Laboratorio III.

Comisión: 2.

Integrantes:Julian Auteri, Lucas Hourcade, Nicolas Nieto.

Resumen:

Nuestra aplicación es un videojuego titulado “Gladiathor the Fool”, en la cual 4 héroes se embarcan a un coliseo en busca de derrotar a todos los enemigos y reclamar la gran recompensa. Es un juego por turnos donde controlas a 4 personajes, cada uno distinto del otro y debes pelear contra enemigos, lo que te dará oro para comprar objetos, para así poder enfrentarte a los siguientes enemigos y así hasta terminar el juego.

Al principio intentamos plantear todo lo posible en el uml, y dividirnos en 2 personas programando y 1 ayudando a ambos buscando información y tirando ideas, poco a poco el uml empezaba a quedar algo desactualizado asi que seguiamos modificandolo para reflejar la realidad y hubieron grandes cambios con respecto a nuestra idea original del juego, mayormente por falta de tiempo, inexperiencia o que nuestra no era posible llevarla a cabo de la forma en que queríamos.

Informe Tecnico:

La clase principal y la cual llama el Main es Coliseo, en esta ocurre la gran mayoría de las situaciones del juego, y es la que unifica a todo el programa. Sus métodos principales “eventos” y “combate” son las que generan que el juego progrese y llegue a un final. Se relaciona con la Tienda para poder comprar objetos que te ayuden a progresar en el juego; todo el package “Criatura” para poder usar personajes y los monstruos en la función de combate; con MensajeGenerico para poder mandar mensajes aleatorios durante el combate; y con MiAyudante para que en el caso de que alguna decisión tomada por el usuario no esté contemplada en el juego se le informe y tenga la opción de volver a intentarlo.

MensajeGenerico es una clase genérica la cual utilizamos en el método del Coliseo “mensajeRandom” para poder enviar cada ciertos turnos un mensaje que podría ser del monstruo a enfrentar o de los personajes jugables.

Tienda es una clase a la que se ingresa por el Coliseo luego de terminar un combate satisfactoriamente. Acá podrás gastar el oro obtenido por los combates en objetos activos o pasivos que ayudarán a los personajes en su aventura. Se relaciona con Objeto en que estos son justamente los que vas a vender; con JsonObjeto para guardar todos los productos de la tienda a partir de un archivo; para que en el caso de que alguna decisión tomada por el usuario no esté contemplada en la tienda se le informe y tenga la opción de volver a intentarlo; y con la Interfaz IMostrarObjeto para poder ver la lista de objetos guardados en la tienda.

Objeto es una clase que se utiliza en Tienda e Inventario y guarda toda la información necesaria de 1 objeto para ambas clases.

JsonObjetos es una clase que se utiliza en Tienda para generar un archivo que al principio se guardaran todos los objetos que irán en la tienda y posteriormente que esta pueda pedir de ese archivo los datos para utilizarlos.

IMostrarObjeto es una interfaz que se utiliza en Tienda e Inventario para mostrar todos los objetos guardados.

Inventario es una clase que se utiliza en Personaje, la cual guarda los objetos (de la clase Objeto) comprados en la tienda entre objetos activos (que podrás usar en el método “combate” de la clase coliseo) y pasivos que aumentan las estadísticas de los personajes permanentemente.

Ser es una clase con la cual el Coliseo genera Monstruos para combatir en el método “combate” y de la que extiende la clase Personaje.

Personaje es una clase abstracta la cual tiene un atributo estático de oro y otro estático de Inventario, para que todos los personajes compartan el mismo oro y objetos. Se relaciona con la clase Ser ya que es hija de esta y le da parte de las estadísticas que tendrán los personajes del jugador; con la clase Pjugable siendo su padre; con Inventario usandola para guardar los objetos que fue comprando en la tienda; y con Coliseo siendo su atributo “jugador” para así poder controlarlos en el método “combate”.

Pjugable: es una clase hija de Personaje la cual tiene métodos con los que “inicializa” los atributos de cada personaje según unos valores predeterminados dependiendo de a qué clase pertenecen. Se relaciona con Coliseo en el método “llenarArrayJugador”

MiAyudante
una clase con métodos únicos donde en cada uno se valida las decisiones del usuario, se relaciona con Coliseo y Tienda para validar lo que el usuario tenía que escribir; tirando excepciones usadas por el sistema y otras personalizadas las cuales son: “CaracterInvalidoExcepcion” y “EleccionEquivocadaExcepcion”; además validamos las acciones que hace en la tienda con: “NoExisteObjetoExcepcion” y “DineroInsuficienteExcepcion”.

CaracterInvalidoExcepcion: se utiliza en mi ayudante y valida si el usuario ingresa una cadena de caracteres.

EleccionEquivocadaExcepcion: cuando el usuario debe tomar decisiones con un número de opciones, si ingresa un número incorrecto de las opciones que existen o este es negativo o igual a cero entonces se tira la excepción.

NoExisteObjetoExcepcion: valida si la tienda contiene el objeto que el usuario ingreso por pantalla.

DineroInsuficienteExcepcion: valida si el usuario no tiene dinero suficiente para comprar un objeto.

CantidadInvalidaExcepcion: valida si compra cantidades válidas de elementos en la tienda.

Manual de Usuario:

Las interfaces ocurren en Tienda y Coliseo, donde en la última la vida y nombre del Monstruo aparece, como también de los personajes, quienes además pueden ver su

ataque. El juego indica de quién es el turno y en caso de ser del monstruo declara si falló el ataque o no, y en caso de que no a quien atacó y cuánto daño le hizo. En caso de ser el turno de alguno de los personajes el menú te mostrará a quien le toca y podrás decidir si atacar, defender o usar un objeto.
Si atacas aparecerá si fallaste el ataque o no y en caso de que no, cuánto daño le hiciste al monstruo.
Si defiendes el juego te avisará que eres inmune al siguiente ataque del monstruo durante esa ronda.
Si deseas usar un objeto activo el juego primero se fijará si tienes o no alguno (en caso de que no te avisa) y luego te pregunta cual deseas usar, en caso de ser un objeto que se use en los personajes te preguntará en quien deseas usarlo.
Si eres derrotado aparecerá en la pantalla un mensaje avisandolo pero si ganaste aparecerá cuánto oro obtuviste y se te preguntará si deseas ir a la tienda, en caso de que si podrás decidir si comprar un objeto activo o pasivo, luego se te mostrará una lista del tipo de objeto seleccionado y podrás escribir el nombre del objeto a comprar, una vez realizada tus compras o declarando no ir a la tienda volverás al siguiente combate.
Una vez derrotados los 10 enemigos recibirás un mensaje declarando que ganaste el juego.

Diario de trabajo:

Semana 1 (desde 27/5):

Creamos el documento para el informe, un excel para la matriz de soluciones, planteamos qué programa vamos a hacer y empezamos a graficarlo en un UML.
Creamos las bases de todas las clases en el proyecto.

Semana 2 (desde 3/6):

Eliminamos que los monstruos se carguen aleatoriamente, ahora lo hacen según el nivel en el que estas.
Creamos las funciones “evento”, crearMonstruo (estas 2 de la clase Coliseo) y gran parte de las funciones de la clase “Tienda”.
Creamos un repositorio de Github para poder trabajar más rápido.
Creamos las estadísticas (son atributos) que tendrían los personajes y monstruos.

Semana 3 (desde 10/6):

Creamos los nombres de los monstruos.

Reestructuramos la clase tienda (ya que no tenía forma de usar el oro).

Creamos el package “Criaturas” para separarlas del juego en sí.

Pusimos las estadísticas base que tendrán cada personaje.

Progresamos en la creación de la función de “combate” y “turnos”.

Creación de las primeras excepciones juntos con la clase “MiAyudante” y sus métodos, además creamos los métodos para la clase Inventario .

Método en “MiAyudante” para validar que en un dato especifico que solo necesita un carácter no ingrese un número.

Método en “MiAyudante” para validar que el usuario ingrese un número de opción válido y no un dato incorrecto.

Semana 4 (desde 17/6):

Terminación del método combate.

Creación de métodos para ordenar por selección para decidir los turnos.

Creación de métodos para que los monstruos decidan a quién atacar.

Creación e implementación del atributo “defender” (de la clase Personaje).

Arreglamos la mayoría de los warnings.

Borramos el sistema de magia.

Borramos la clase “Monstruo” (estos ahora se crean en la clase “Ser)

Creamos la clase “Pjugable”.

Semana 5 (desde 24/6):

Ajustes estadísticas de objetos, inventario y personajes.

Cambios en el uso de los objetos activos.

Creación clase “JsonObjetos” y sus métodos.

Implementación de “JsonObjetos” en la clase “Tienda”, creación de métodos en esta para usarlos.

Metodo en “MI Ayudante” para validar si la cantidad que el usuario quiere comprar no sea negativa.

Creación del boss final.

Matriz de soluciones

Problemas	Soluciones
Cargar de forma randomizada cada monstruo, implicaba muchas variables sujetas a error entorpeciendo el flujo del código	Eliminación de carga de monstruos random
Por como está diseñado el juego, el monstruo en la primera posición del ArrayList jamás aparecia	Creamos el monstruo Easter Egg en la posición 0
En el HashSet no podíamos sacar los objetos específicamente, algo que nuestro código requería	Cambios de HashSet por HashMap en ObjetosActivos y ObjetosPasivos
No había forma de comprar los objetos	Implementación del atributo "oro"
No teníamos forma de saber cuántos Objetos Activos habías comprado	Creación de atributo cantidad
Al estar todas las clases en el mismo package, Coliseo podía acceder a las stats de los monstruos y los jugadores sin necesidad de getters y setters	Creación package Criatura
No sabíamos cómo decidir los turnos	Creamos un ArrayList llamado Combatientes que se ordenaba por el método de selección para decidir turnos

Al estar en dos arraylist diferentes (combatientes para los turnos y jugadores), no podíamos encontrar donde estaba el personaje en el ArrayList jugadores (esto hacía que no podamos pegar por ejemplo)	Creamos un bucle While para "matchear" los dos arrays
No había forma de saber a quién iba a atacar el enemigo	creación de "ia" de los monstruo (función para decidir a quién atacar)
Los menúes no eran "aprueba de tontos"	Creación de CaracterInvalidoExcepcion (para que el usuario no ingrese un numero si no se lo pide)
No teníamos forma de saber cuántos y qué objetos comprabas, qué estadísticas se modificaban y que precio tenían estos	creación de la clase Objeto donde se guardaban estos datos y creación de dos hashmap para guardar los objetos en la tienda (se separan por activos y pasivos)
Si el objeto ingresado no estaba en la tienda porque, por ejemplo, lo habías escrito mal, perdías la oportunidad de comprar y saltaba al siguiente combate	Creacion de NoExisteObjetoException
Similar al problema anterior, si querías usar algún objeto que no tenias, se saltaba tu turno y no había forma de saber por qué	Creación de otro método que implementa NoExisteObjetoException
En los distintos menús si elegías un número distinto a las opciones existentes se rompía	Creación de EleccionEquivocadaExcepcion
El usuario no tenía forma de saber qué pasaba si no tenias dinero suficiente para comprar un objeto	Creación de DineroInsuficienteException
Las excepciones no solucionan todos los problemas, simplemente hacían que el código no se rompiera	Creación de bucles do - while para darle la oportunidad al usuario de ingresar un dato correcto

No encontramos forma posible de implementar la genericidad y la magia, entorpecia y rompía el código	Borrar sistema de magia
Al borrar la magia el tema "Genericidad" no se implementaba en ningún momento	Creación de un sistema de mensajes aleatorios que recibía números o letras y en base a eso lanzaba comentarios dichos por los personajes o los monstruos
Al borrar la magia, las clases Monstruo, Arquero, Guerrero, Mago y Clérigo carecían de sentido	Eliminación de clase Monstruo (ahora son seres) y creación de la clase Pjugable para los personajes que manipula el usuario (estos seguían teniendo una clase aparte debido al inventario)
La diferencia entre objetos pasivos y activos, es que los activos modifican una stat únicamente durante el combate en cuestión, luego volverían a la normalidad, esto no estaba sucediendo	Cambios en el uso de los objetos activos
Los objetos tenían que ser creados obligatoriamente cada vez que se iniciaba el juego	Creación clase “JsonObjetos” y sus métodos e implementación en tienda
nos dimos cuenta que tener dos hashmap con el mismo tipo de objeto no es una buena práctica	Guardamos todo en un Hashmap y le aplicamos atributos únicos a los objetos para poder identificarlos
la clase Json solo tenía como metodo recibir un hashmap para poder guardarlo o descargarlo y eso bloqueaba la flexibilidad para otros tipos de datos	Los métodos de Json son estáticos y solo utilizan un archivo para abrirlo o guardarlo , otras clases se encargará de implementar sus métodos para ver cómo abrirlo o guardarlo a partir de sus datos específicos.

Fuentes consultadas:

Bibliografía otorgada por la universidad en el campus de nuestra comisión.

https://elcodigoascii.com.ar/
https://www.scaler.com/topics/int-to-char-in-java/
https://stackoverflow.com/questions/1401481/how-to-do-a-system-pause-in-java-for-debugging