Félix San José Hernán

Programación
DAW - 2023/2024

JUEGO DE SIMULACIÓN DE COMBATES ENTRE BANDOS AMBIENTADO EN

IES María Moliner

Fecha: 19/03/2024

Contenido

Contenido 2
Introducción al proyecto 3
Objetivos 3
1. Estructura del proyecto 3
2. Creación de clases 3
3. Diseño de combate 3
4. Integración de la lógica 4
5. Pruebas de integración y funcionales 4
6. Ajustes Finales 4
7. Documentación 4
Método de desarrollo del proyecto 5
Relación entre clases 6
8. Clase Personaje (Abstracta): 6
8.1 Métodos: 6
9. Clase Guerrero (Subclase de Personaje): 7
9.1 Métodos: 7
10. Clase Mago (Subclase de Personaje): 7
10.1 Métodos: 7
Programa principal 8
Tratamiento de errores 11
Principios de la Programación Orientada a Objetos utilizados: 12
Pruebas Unitarias con JUnit 13
Posibles mejoras y ampliaciones 14
Bibliografía 15

Introducción al proyecto

Este simulador de combate es un proyecto en Java que simula un combate por turnos entre dos bandos enfrentados ambientado en el universo de World of Warcraft. Cada bando está compuesto por cuatro personajes del tipo mago o guerrero con puntos de vida, puntos de ataque y defensa. El objetivo del juego es derrotar a todos los personajes del equipo oponente.

El proyecto se ejecuta y muestra al usuario haciendo uso de un entorno de desarrollo integrado, permitiendo interactuar mediante la consola de este introduciendo las distintas acciones a realizar.

Objetivos

Con las instrucciones aportadas por el profesor sobre la realización del proyecto se establecieron los siguientes objetivos del proyecto.

1. Estructura del proyecto

Se establece la arquitectura general del proyecto, identificando las clases, sus métodos y atributos, así como las relaciones entre ellas para lograr el resultado deseado y establecido.

2. Creación de clases

Implica diseñar e implementar las clases necesarias para representar a los personajes del juego en el simulador de batallas. Esto incluye definir las características y comportamientos específicos de cada tipo de personaje, como guerreros y magos.

3. Diseño de combate

Creación de un sistema de combate interactivo por turnos para el juego. Este objetivo implica diseñar y desarrollar la lógica detrás de las interacciones entre los personajes durante las batallas, incluyendo el cálculo de daño, la aplicación de habilidades especiales y la gestión de turnos.

4. Integración de la lógica

Integrar la lógica del juego, incluyendo la estructura del proyecto, las clases de personajes y la lógica de combate, con la interfaz de usuario por consola. Esto implica conectar la lógica subyacente del juego con los elementos visuales de la interfaz de usuario, asegurando una experiencia de juego cohesiva y fluida para los jugadores.

5. Pruebas de integración y funcionales

Realizar pruebas exhaustivas para verificar la integración correcta entre la lógica del juego y la interfaz de usuario, así como para garantizar el correcto funcionamiento de todas las características y funcionalidades del juego. Esto incluye pruebas de flujo de juego, pruebas de interfaz de usuario y pruebas de casos de uso específicos para asegurar que el juego cumpla con los requisitos y expectativas establecidos.

6. Ajustes Finales

Realizar ajustes finales en el proyecto para optimizar el rendimiento, mejorar la jugabilidad y corregir posibles errores o problemas identificados durante las pruebas. Esto puede incluir optimizar el código, ajustar el equilibrio de juego, mejorar la interfaz de usuario y agregar nuevas características o mejoras basadas en los comentarios recibidos.

7. Documentación

Crear una documentación detallada del proyecto que describa la arquitectura general, la estructura del código, las decisiones de diseño, las instrucciones de instalación y uso, así como cualquier otra información relevante para facilitar el mantenimiento y la comprensión del proyecto por parte de otros desarrolladores o usuarios.

Método de desarrollo del proyecto

Para la realización del proyecto se ha utilizado como entorno de desarrollo integrado Apache Netbeans IDE 20. Este IDE proporciona un conjunto de herramientas robustas y eficientes que facilitan la codificación, depuración y gestión del proyecto. Con su interfaz intuitiva y sus características avanzadas, como la finalización de código automática y el resaltado de sintaxis, se agilizó el proceso de desarrollo.

Desde Netbeans, la creación del proyecto se ha realizado utilizando Java Maven como herramienta de gestión de dependencias y construcción de proyectos. Maven simplificó la gestión de bibliotecas externas y la compilación del código, asegurando que todas las dependencias estuvieran correctamente configuradas y actualizadas.

Relación entre clases

En el proyecto las clases están diseñadas siguiendo un enfoque de programación orientada a objetos, lo que permite una representación eficiente de los diferentes tipos de personajes y sus relaciones. A continuación, se detallan las relaciones entre las clases junto con algunos aspectos técnicos:

8. Clase Personaje (Abstracta):

La clase abstracta Personaje define la estructura base común para todos los personajes en el juego. Contiene atributos como nombre, puntos de vida (hp), puntos de ataque (atk), puntos de defensa (def), entre otros. Además, incluye métodos como atacar(), defender() y habilidad(), siendo este último abstracto. La clase contiene también los Getter y Setter de cada atributo y por defecto la vida de cada personaje se establece a 100, valor que se determina con la constante SALUD_INICIAL.

8.1 Métodos:

-atacar():

Este método devuelve un número de tipo int aleatorio comprendido entre el máximo ataque del personaje y la mitad del máximo (100 de ataque -> máximo 100, mínimo 50).

-defender():

Este método devuelve un número de tipo int aleatorio comprendido entre la máximo defensa del personaje y la mitad del máximo (100 de defensa -> máximo 100, mínimo 50).

-habilidad():

Método abstracto que devuelve un int, este método será sobrescrito en las clases mago y guerrero, cuyos métodos funcionan de manera distinta.

La clase Personaje utiliza el concepto de abstracción para definir una plantilla genérica para los personajes, sin proporcionar una implementación específica de ciertos métodos. Esto permite la creación de subclases que pueden personalizar el comportamiento según sus características únicas.

9. Clase Guerrero (Subclase de Personaje):

La clase Guerrero extiende la clase Personaje y agrega funcionalidades específicas para los personajes que se especializan en el combate cuerpo a cuerpo. Implementa el método abstracto habilidad() definido en la clase Personaje para adaptarlo a la clase guerrero.

La relación de herencia entre Guerrero y Personaje permite que la clase Guerrero herede los atributos y métodos de la clase Personaje, lo que evita la duplicación de código y promueve la reutilización.

9.1 Métodos:

-habilidad():

Mediante un número aleatorio se determina la probabilidad de realizar un ataque que haga el doble de daño con una probabilidad del 15%, mostrando por pantalla el acierto. En caso de no acertar, el método devolverá 0 y mostrará por pantalla un mensaje.

10. Clase Mago (Subclase de Personaje):

La clase Mago también extiende la clase Personaje, pero se enfoca en los personajes que utilizan habilidades mágicas y hechizos en combate. Implementa los métodos de la clase Personaje al igual que Guerrero, además de un método curar() propio único en esta clase.

La clase Mago hace uso del polimorfismo al proporcionar implementaciones únicas de los métodos abstractos heredados de Personaje. Esto permite que los objetos de tipo Mago se comporten de manera diferente a los objetos de otras subclases de Personaje, como Guerrero.

10.1 Métodos:

-habilidad():

Parecido a su funcionalidad en la clase de guerrero, mediante un número aleatorio se determina el acierto de la habilidad, con una probabilidad del 25% tendrá éxito y devolverá el máximo daño de ataque que tenga el personaje, evitando así la aleatorización de daño que se realiza al ejecutar el método atacar().

-curar():

Este método permitirá al usuario recuperar 10 puntos de vida al personaje con una probabilidad del 30%. En caso de que el personaje ya tenga 100 de vida no podrá sumar más vida, si la suma de la vida actual más los 10 puntos de vida supera los 100 puntos se establecerá la vida en 100. El método mostrará por pantalla mensajes cuando surta efecto y en caso de fallar.

En resumen, la estructura de clases en el proyecto se basa en la herencia y la abstracción para modelar eficazmente las características y el comportamiento de los diferentes tipos de personajes en el juego. Esta arquitectura facilita la escalabilidad y el mantenimiento del código, al tiempo que proporciona una representación clara y modular de los elementos del juego.

Programa principal

Aquí se definen métodos para manejar la lógica del juego y se gestionan las interacciones con el usuario y el ordenador.

Métodos Principales:

1. imprimirFaccion(Personaje[] faccion):

Este método imprime los detalles de los personajes de una facción dada, como el nombre, la clase, el ataque y la defensa.

2. elegirFaccion(Personaje[] ali, Personaje[] horda):

Permite al usuario elegir un bando para luchar (Alianza o Horda) mediante la entrada de datos. Retorna el número correspondiente al bando elegido.

3. elegirPersonaje(Personaje[] faccion, String accion):

Permite al usuario elegir un personaje de una facción dada. Retorna el número correspondiente al personaje elegido.

4. fin(int contMuertesJugador, int contMuertesOrdenador, boolean ganador, String bandoJugador, String bandoOrdenador):

Imprime el resultado del juego, indicando si el jugador ganó, perdió o hubo un empate.

5. accion(int respuesta, Personaje pj):

Permite a un personaje realizar una de varias acciones basadas en la elección del usuario: atacar, defender, usar una habilidad especial o curarse. La acción se especifica mediante el parámetro respuesta y se aplica al personaje pj. Dependiendo de la acción elegida:

Atacar (1): pj realiza un ataque, generando puntos de ataque y mostrando un mensaje con los puntos generados.

Defender (2): pj se defiende, generando puntos de defensa y mostrando un mensaje con los puntos generados.

Usar habilidad (3): pj usa una habilidad especial, generando puntos que se asignan pero sin mostrar mensaje.

Curarse (4): Si pj es un Mago, se cura a sí mismo, sin generar puntos.

El método retorna los puntos generados por la acción, excepto en el caso de curación, donde retorna cero.

6. main(String[] args):

Aquí se orquesta el flujo del juego, gestionando los turnos del jugador y del ordenador, así como el desarrollo del combate, la selección de personajes y la determinación del ganador.

Para interactuar con el usuario, se utiliza la clase Scanner para la entrada de datos, permitiendo al usuario tomar decisiones durante el juego. Además, se emplean métodos de impresión en consola para mostrar mensajes y detalles relevantes del juego, manteniendo al usuario informado en todo momento.

Para manejar posibles errores de entrada de datos por parte del usuario, se implementa el manejo de excepciones, lo que permite capturar estos errores y proporcionar mensajes claros al usuario, facilitando la corrección de los mismos.

Durante el combate, se generan respuestas aleatorias del ordenador utilizando la función Math.random(), lo que añade un componente de imprevisibilidad al juego y aumenta su emoción.

Se instancian objetos de las clases Guerrero y Mago para representar a los personajes del juego. Estas clases proporcionan métodos específicos, como atacar(), defender(), habilidad() y curar(), que se utilizan para ejecutar las acciones durante el combate, agregando diversidad y estrategia al juego.

Para gestionar las acciones realizadas durante el combate, se consideran las siguientes reglas:

- Ataque vs Defensa: Si un personaje ataca y el otro defiende, se compara el ataque con la defensa. Si la defensa es mayor o igual, el ataque se neutraliza; si no, se reduce la salud del defensor por la diferencia.

- Defensa vs Ataque: Similar al caso anterior pero invirtiendo los roles. Si la defensa supera al ataque, se neutraliza; de lo contrario, se reduce la salud del atacante.

- Ambos Defienden: No ocurre daño, y se muestra un mensaje indicando que ambos intentaron bloquear.

- Habilidad vs Defensa: Si un personaje usa una habilidad y el otro defiende, la defensa no surte efecto y la habilidad realiza el daño completo.

- Ambos Atacan o Usan Habilidad: Si ambos personajes optan por atacar o usar habilidades, ambos sufren daño directo sin consideraciones defensivas, reduciendo su salud por el valor del ataque o habilidad del oponente.

- Curación: Si algún personaje intenta curarse, simplemente se muestra un mensaje indicativo.

Cuando los puntos de vida de un personaje del jugador se reducen a 0, el contador de muertes del jugador aumenta en 1 y el usuario debe elegir un nuevo personaje vivo para continuar la batalla. Si un personaje del ordenador muere, su contador de muertes aumenta en 1 y se selecciona un nuevo personaje vivo de manera aleatoria.

El juego continúa hasta que uno de los contadores de muertes alcanza el valor de 4, momento en el que se ejecuta el método fin() para determinar el resultado final del juego.

Tratamiento de errores

Se hace uso del manejo de errores para garantizar una experiencia de usuario fluida y evitar fallos inesperados durante la ejecución del programa.

InputMismatchException:

Esta excepción se utiliza para manejar errores cuando se espera un tipo de dato específico del usuario (por ejemplo, un entero) y se proporciona un tipo de dato diferente.

Se captura esta excepción al leer la entrada del usuario utilizando el objeto Scanner. Si se detecta un tipo de dato incorrecto, se muestra un mensaje descriptivo al usuario indicando el error y se le pide que ingrese el tipo de dato correcto.

OpcionNoValidaException:

Esta excepción personalizada se utiliza para manejar casos donde el usuario elige una opción inválida durante el juego, como seleccionar una opción de menú no disponible.

Se lanza esta excepción cuando se detecta una opción inválida, como elegir un personaje que ya está muerto o seleccionar una acción no permitida en el juego en ese momento.

Se muestra un mensaje claro y descriptivo al usuario, explicando el error y proporcionando orientación sobre cómo corregirlo.

Propagación de excepciones:

Cuando se captura una excepción, se gestiona adecuadamente mostrando un mensaje informativo al usuario, pero también se propaga la excepción si es necesario para que el programa no continúe ejecutándose en un estado inconsistente.

Esto garantiza que los errores sean manejados en el nivel adecuado y que el programa pueda continuar ejecutándose de manera coherente o, en casos extremos, finalizar sin causar daños adicionales.

Principios de la Programación Orientada a Objetos utilizados:

En este proyecto, se aplican varios principios fundamentales de la Programación Orientada a Objetos (POO) para garantizar un diseño estructurado y modular:

Herencia: Se utiliza la herencia para modelar la relación "es un" entre las clases Mago y Guerrero y la clase abstracta Personaje. Esto permite definir un conjunto común de atributos y comportamientos en la clase Personaje, que luego se especializan en las subclases Mago y Guerrero. Esta jerarquía de clases facilita la reutilización del código y promueve una estructura de código más organizada y mantenible.

Encapsulación: Se aplican los conceptos de encapsulación al definir los campos de las clases como private o protected según corresponda. Esto asegura que los datos estén protegidos dentro de las clases y solo se puedan acceder y modificar mediante métodos específicos. Por ejemplo, los atributos de los personajes se encapsulan y se acceden a través de métodos de acceso (getters y setters), lo que proporciona un mayor control sobre el estado interno de los objetos y evita la manipulación directa de los datos.

Abstracción: Se utiliza la abstracción al definir la clase abstracta Personaje, que encapsula las características comunes y el comportamiento general de los personajes del juego. Esto permite definir un conjunto de métodos abstractos que deben implementarse en las subclases concretas (Mago y Guerrero), lo que fomenta la cohesión y la modularidad del código al centrarse en los aspectos esenciales del problema sin preocuparse por los detalles de implementación.

Polimorfismo: Se aprovecha el polimorfismo al permitir que los objetos de las subclases (Mago y Guerrero) se comporten como objetos de la clase base (Personaje). Esto se logra al llamar a los métodos polimórficos como atacar(), defender() y habilidad() a través de referencias de la clase base, lo que permite un tratamiento uniforme de los diferentes tipos de personajes en el juego.

Pruebas Unitarias con JUnit

Este proyecto incluye una serie de pruebas unitarias implementadas utilizando el framework de pruebas JUnit. Estas pruebas están diseñadas para verificar el funcionamiento correcto de las clases Mago y Guerrero, así como de sus métodos asociados.

La clase testPersonaje contiene una serie de métodos de prueba, cada uno etiquetado con la anotación @Test, que evalúan aspectos específicos del comportamiento de las clases Mago y Guerrero. Estos métodos incluyen:

testConstructorMago(): Verifica que se construya correctamente un objeto de tipo Mago.

testConstructorGuerrero(): Comprueba que se construya correctamente un objeto de tipo Guerrero.

testAtacarMago(): Asegura que el método atacar() de un Mago devuelva un valor mayor que cero.

testDefenderGuerrero(): Verifica que el método defender() de un Guerrero devuelva un valor mayor que cero.

testHabilidadMago() y testHabilidadGuerrero(): Comprueban que los métodos habilidad() de ambas clases no devuelvan null.

testCurarMago(): Verifica que el método curar() de un Mago aumente la salud del personaje a 100 si está por debajo de ese valor.

Antes de cada prueba, se ejecuta el método configuración(), etiquetado con @BeforeEach, que inicializa los objetos Mago y Guerrero necesarios para las pruebas. Esto garantiza un entorno limpio y consistente para cada caso de prueba.

Posibles mejoras y ampliaciones

Implementación de interfaz gráfica de usuario (GUI): Actualmente el programa funciona en la consola, pero podría mejorarse considerablemente agregando una interfaz gráfica de usuario para hacerlo más interactivo y visualmente atractivo.

Personalización de equipos y variedad de personajes: Permitir al jugador personalizar su equipo eligiendo entre una mayor variedad de personajes con habilidades y estadísticas distintas antes de entrar en batalla, agregaría un elemento adicional de personalización y estrategia al juego.

Implementación de inteligencia artificial avanzada: Mejorar la lógica de la IA para que tome decisiones más inteligentes y estratégicas durante el combate, aumentando así el desafío para el jugador.

Optimización del rendimiento: Identificar y corregir posibles cuellos de botella o áreas de ineficiencia en el código para mejorar el rendimiento del programa, especialmente en juegos más grandes o en sistemas con recursos limitados.

Soporte para multijugador: Permitir partidas multijugador local para que los jugadores puedan enfrentarse entre sí, agregando una nueva dimensión al juego y fomentando la competencia entre amigos.

Añadir más modos de juego: Introducir diferentes modos de juego que ofrezcan variedad y desafíos únicos para los jugadores. Por ejemplo, se podrían añadir modos de juego como "Supervivencia", donde los jugadores deben enfrentarse a oleadas cada vez más difíciles de enemigos hasta que sean derrotados, o modos 1vs1, 2vs2, 3vs3… ya que la actual manera de jugar es únicamente 4vs4.

Bibliografía

Icono de World of Warcraft:

https://www.cleanpng.com/png-world-of-warcraft-logo-blizzard-entertainment-priv-6340930/

Icono de Netbeans:

https://pbs.twimg.com/media/Dp3nCCdXgAEIMxM.png