Diseño de Software

Proyecto Final

Alumnos: Joaquín Bazterrica, Pablo Martos, Lucas Saeta, Alejandro Sanchidrian, Alejandro Escudero & Guillermo Alvarez

Docente: Miguel Angel Mesas Uzal

Curso: INSO2A

Fecha de entrega: 13/05

Año lectivo: 2022

ÍNDICE

I. INTRODUCCIÓN

Esta será la memoria en la que plasmamos todos los componentes de nuestro trabajo, además de una explicación que ayude a entender el por que de cada aplicación que les hemos dado, usando tanto tablas como diagramas para que cada clase quede lo más clara posible y ayudarnos a presentar este proyecto.

II. MANUAL DE USO

2.1 Atributos

Cuando comienza el juego, puedes elegir entre 5 clases diferentes y cada una va a tener unos atributos.

	POWER	STRENGTH	SHIELD	WISDOM	AGILITY
FIGHTER	15	80	10	5	5
ASSASSIN	20	40	2	5	10
TANK	10	100	15	2	1
MAGE	15	65	1	13	5
NONAME	4	50	4	4	4

2.2 Estados

El personaje y los enemigos van a ir pasando por una serie de estados que van a condicionar el combate.

- Cuando el personaje y los enemigos no estén afectados por ningún estado negativo estarán en **estado normal (estado por defecto).**
- Cuando el personaje haya recibido algún stun, se encontrará en **estado tuneado** y perderá un turno**.**
- Cuando el personaje esté afectado por el estado de **weakened**, su defensa bajará a 0 puntos.

- Cuando el enemigo tenga el estado **untargetable**, no se le podrá atacar.

2.3 Jugador

El jugador va a poder realizar 3 acciones diferentes:

- Atacar: Inflige daño a su enemigo dependiendo de su poder.
- Defenderse: Gana 10 de escudo, disminuyendo el daño recibido.
- Curarse: Gana 20 puntos de Strength.

Cuando el jugador completa el nivel 1, recibe la espada de fuego que aumenta el power del personaje en 10 puntos.

Cuando el jugador completa el nivel 2, recibe el casco legendario que aumenta el shield en 10 puntos.

2.4 Enemigos

Los enemigos van a realizar una serie de acciones que van a depender de cómo vaya el transcurso del combate.

Podrán atacar, defenderse, efectuar una estrategia ofensiva o una estrategia defensiva.

	POWER	STRENGTH	SHIELD	DEFAULT STRATEGY
BEAST	15	30	2	OFFENSIVE
SORCERER	15	20	5	DEFENSIVE
WARRIOR	15	25	10	OFFENSIVE

Cómo va a haber 3 niveles los enemigos van a tener multiplicadores de sus atributos por cada nivel, cuanto más grande sea el nivel, más grande el multiplicador, por lo que mejores atributos tendrán los enemigos.

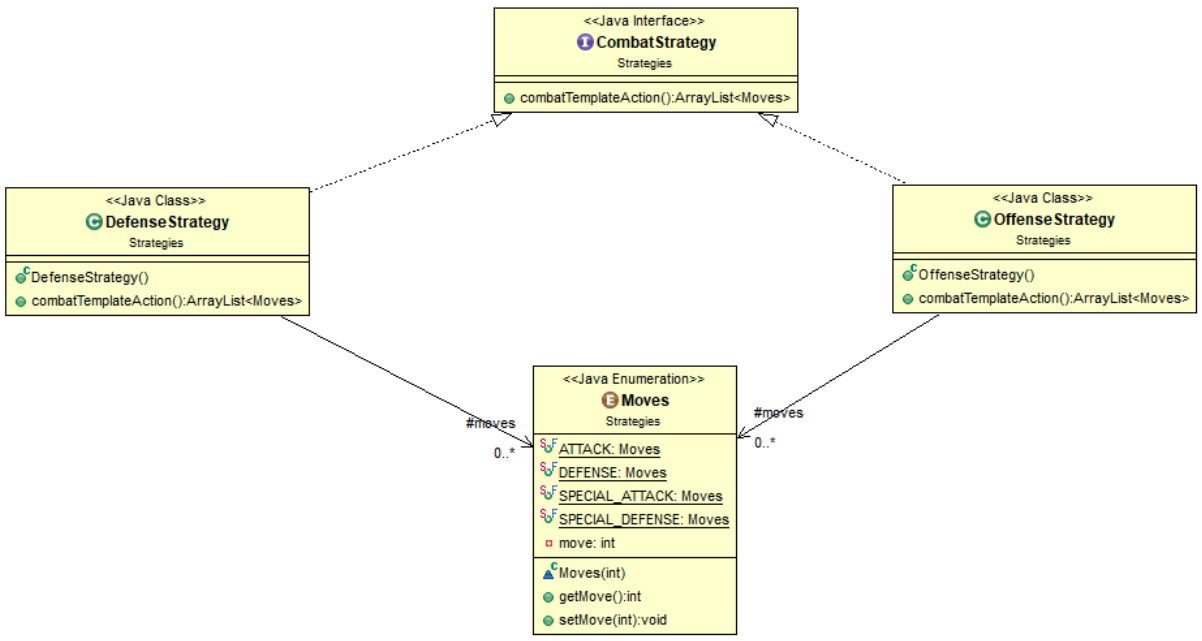
2.5 Mundos

En el juego va a haber 3 mundos:

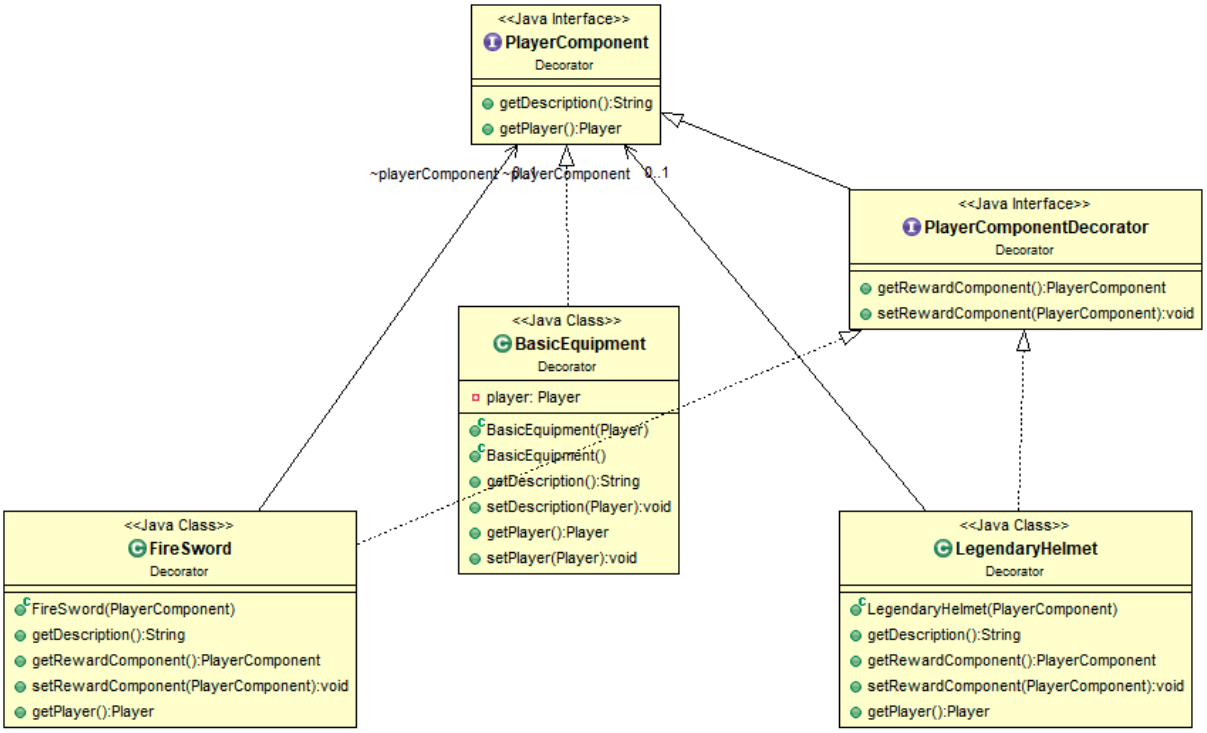
- **Everfrost**: Va a tener un multiplicador de x1 que afectará a los atributos de los enemigos.
- **Forgotten Mountain**: Va a tener un multiplicador de x1.5 que afectará a los atributos de los enemigos.
- **Ruined Castle**: Va a tener un multiplicador de x3 que afectará a los atributos de los enemigos.

III. DIAGRAMA DE CLASES

3.1 Strategy Pattern

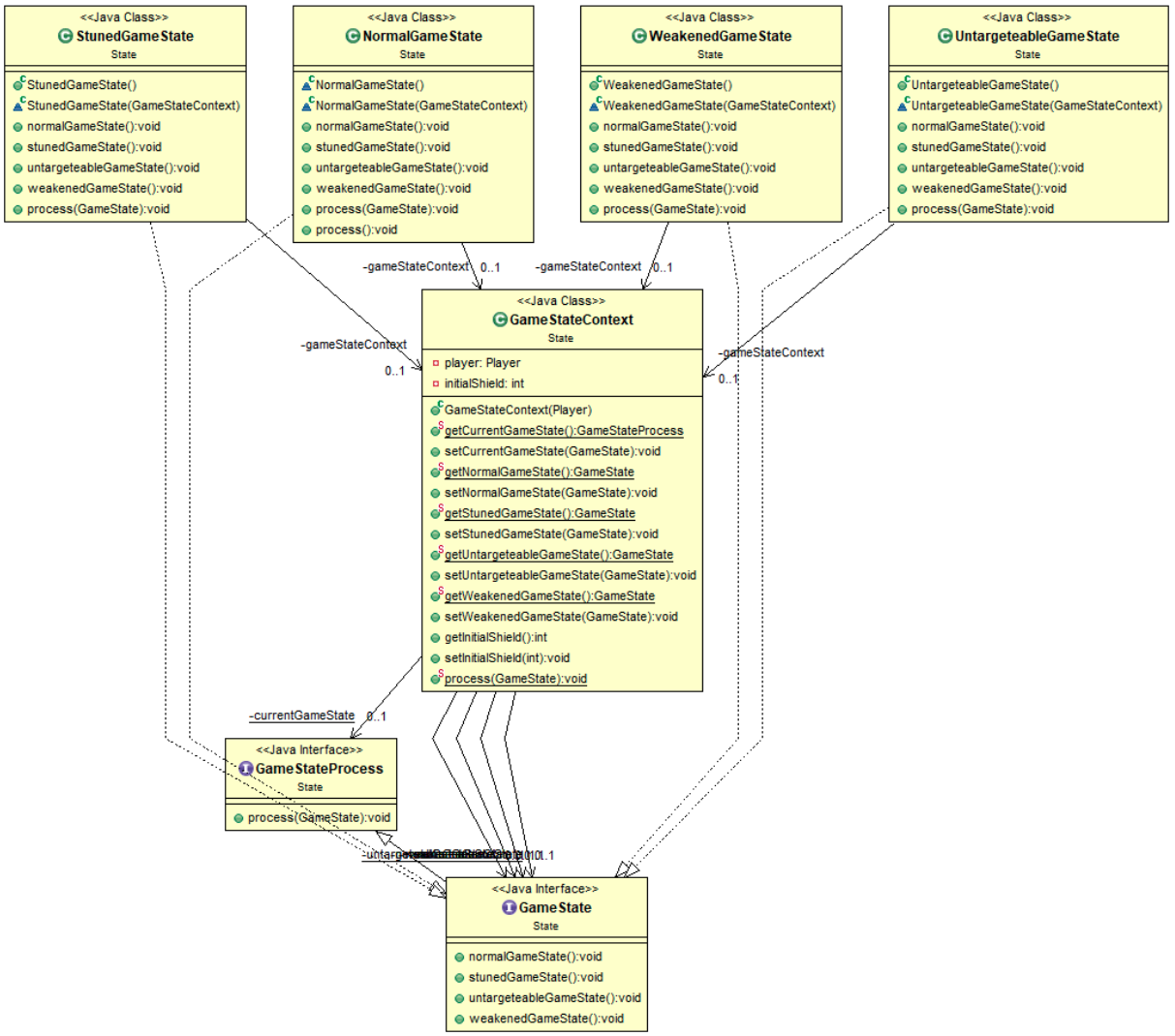


3.1 Decorator Pattern

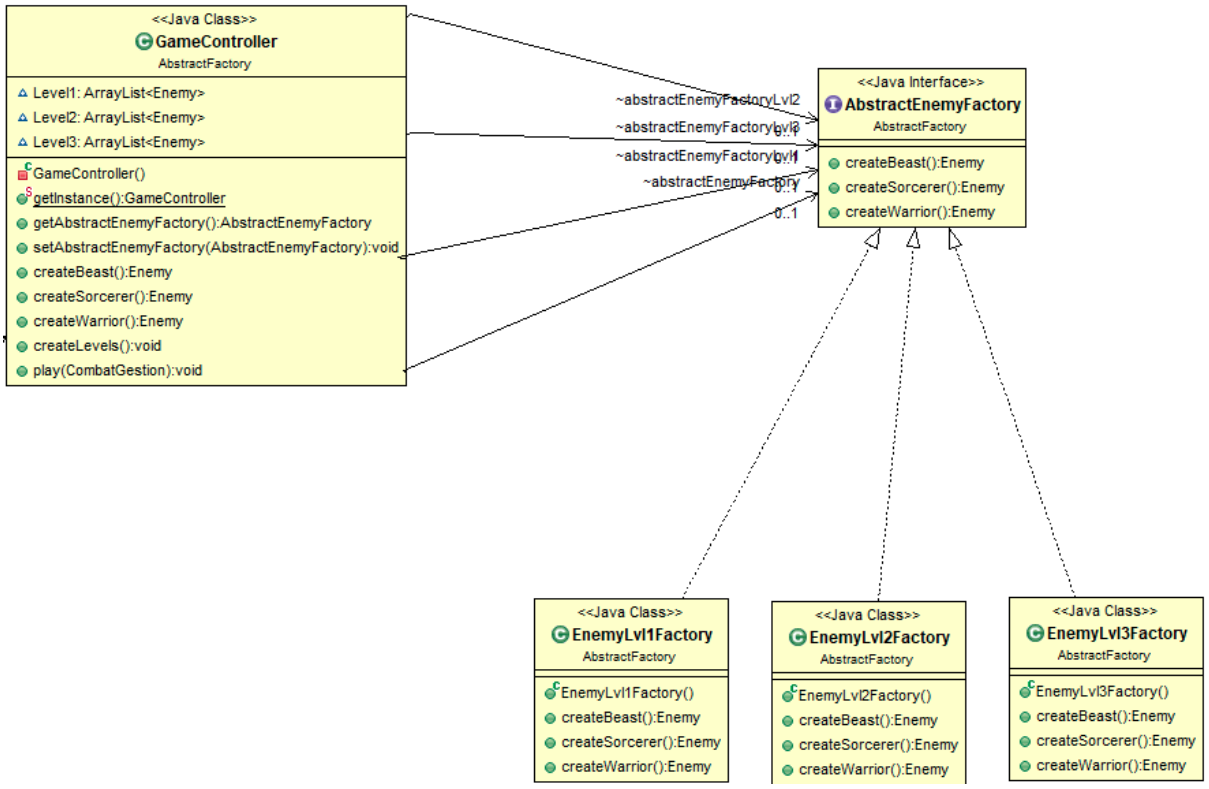


Para crear los diferentes artilugios que va a ir consiguiendo el jugador a lo largo del juego, hemos implementado el patrón decorador. La clase jugador va a importar la base inicial que es BasicEquipment que a través de la interfaz PlayerComponent va a tener los atributos iniciales del jugador. La interfaz PlayerComponentDecorator (que extiende de PlayerComponent) va a ser la encargada de ir añadiendo clases a la base inicial. Estas clases como FireSword o LegendaryHelmet, van a proporcionar nuevos atributos a la clase jugador.

3.3 State Pattern



3.4 Abstract Factory Pattern



Para crear los diferentes enemigos hemos utilizado el patrón Abstract Factory. Hay 3 mundos y 3 enemigos distintos, beast, sorcerer y warrior. Para empezar, creamos una interfaz común con los métodos para crear estos enemigos.

Creamos después 3 clases como factorías para los enemigos de distintos mundos, cada una con sus estadísticas.

Por último, encontramos la clase ”GameController”. Esta es la que manejará el funcionamiento del juego. Para empezar, tiene 3 ArrayList para poder guardar los enemigos de distintos mundos y las factorías para poder crearlos. Una vez empieza el juego, se llama al método “play”. Este, antes que nada, crea todos los enemigos de los mundos con el método “createLevels”, y una vez creados, mientras la vida del jugador sea mayor que 0 y no se haya pasado el juego, iniciamos un bucle para representar los combates.

En el momento que todos los enemigos de un mismo mundo son derrotados, se añade un componente decorador al personaje y se pasa al siguiente nivel.

3.5 Singleton Pattern

Proporcionaremos a GameController una instancia para que los demás usuarios puedan hacer referencia a ella de una forma sencilla con un getinstance() como se ve en este código.

```
private static GameController gameController = new GameController();
```

```
public void startGame(Player player)
{
    GameController.getInstance().play(new CombatGestion(player, new Warrior3()));
}
```

3.6 Template Method Pattern

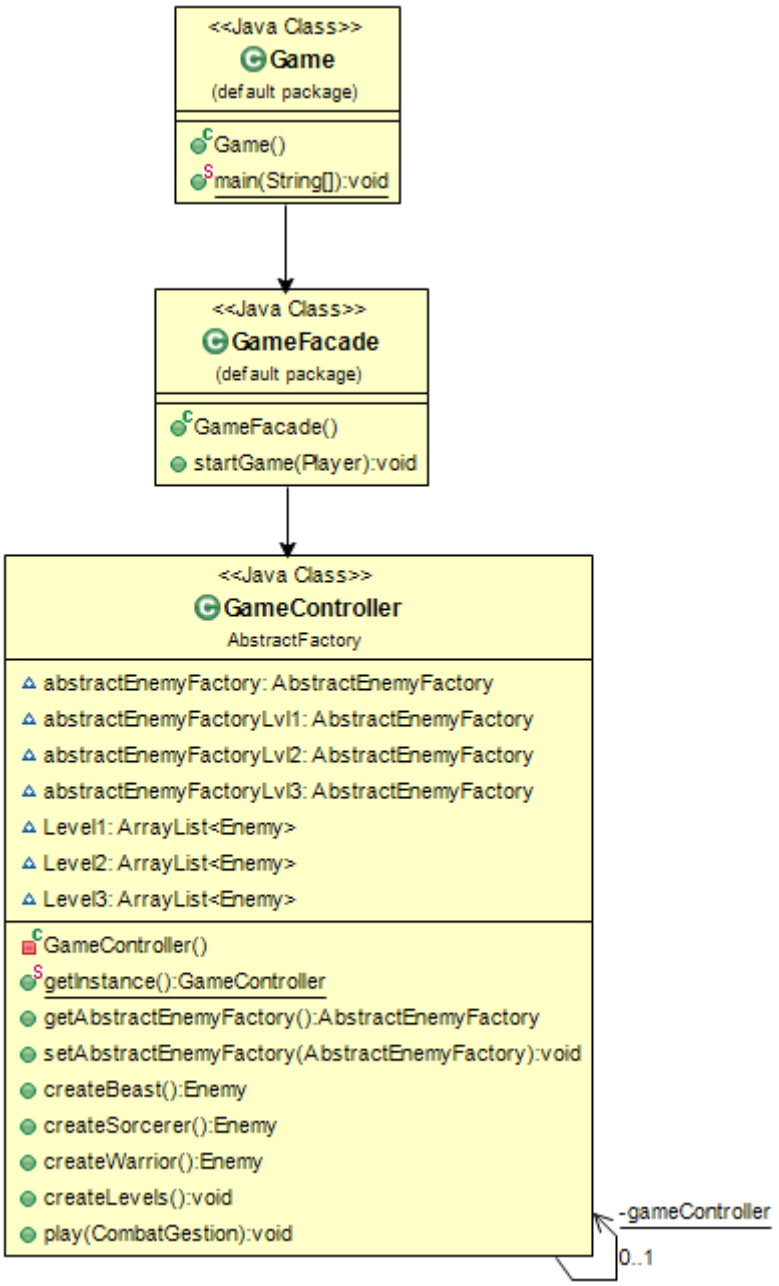
Será la plantilla que usaremos para poner patrones establecidos en la selección de los atributos que tendrá el caso en cuestión, de esta forma serán más fáciles de implementar y tendrán todos una estructura similar aunque sus cualidades sean diferentes.

El caso que se muestra en la captura define el tipo de movimientos que los jugadores serán capaces de hacer, y como se ve la estructura en el ataque y en la defensa es la misma.

```
public ArrayList<Moves> combatTemplateAction()
{
    moves.add(Moves.DEFENSE);
    moves.add(Moves.SPECIAL_DEFENSE);
    moves.add(Moves.ATTACK);

    return moves;
}
```

```
protected ArrayList<Moves> moves = new ArrayList <Moves>();

public ArrayList<Moves> combatTemplateAction()
{
    moves.add(Moves.ATTACK);
    moves.add(Moves.SPECIAL_ATTACK);
    moves.add(Moves.ATTACK);

    return moves;
}
```

3.7 Facade Pattern



Tal y como dice el nombre el patrón Facade es la fachada que será vista por el usuario escondiendo todas las acciones complejas que no le incumben, como por ejemplo la elección del jugador, que será vista por el usuario .

En este caso la hemos implementado en la clase GameFacade que llamara al play() del GameController en donde hacemos referencia mediante el singleton a todas esas complejas acciones de las que hemos hablado antes.

```
        {
        case 1:
            player = new Player(PlayerClass.FIGHTER);
            break;
        case 2:
            player = new Player(PlayerClass.ASSASSIN);
            break;
        case 3:
            player = new Player(PlayerClass.TANK);
            break;
        case 4:
            player = new Player(PlayerClass.MAGE);
            break;
        case 5:
            player = new Player(PlayerClass.NONAME);
            break;
        default:
            System.out.println("La clase escogida no es valida, vuelve a intentarlo");
        }
    }while(clasePlayer < 1 && clasePlayer > 5);

    gameFacade.startGame(player);
}
```

```
import AbstractFactory.GameController;

public class GameFacade
{
    public void startGame(Player player)
    {
        GameController.getInstance().play(new CombatGestion(player, new Warrior3()));
    }
}
```

```
 public CombatGestion(Player player, Enemy enemy)
 {
     this.player = player;
     this.enemy = enemy;
     turns = 0;
     playerAction = 0;
     enemyActions = this.enemy.getMoves();
     gameStateContext = new GameStateContext(player);
 }
```