Grupo BoomBers

INGENIERÍA DEL SOFTWARE

Curso 2024- 2025

Paula Tapias, Eneko Uresandi, Mario Echavarri, Alvaro Blasco y Aitzol Rivera

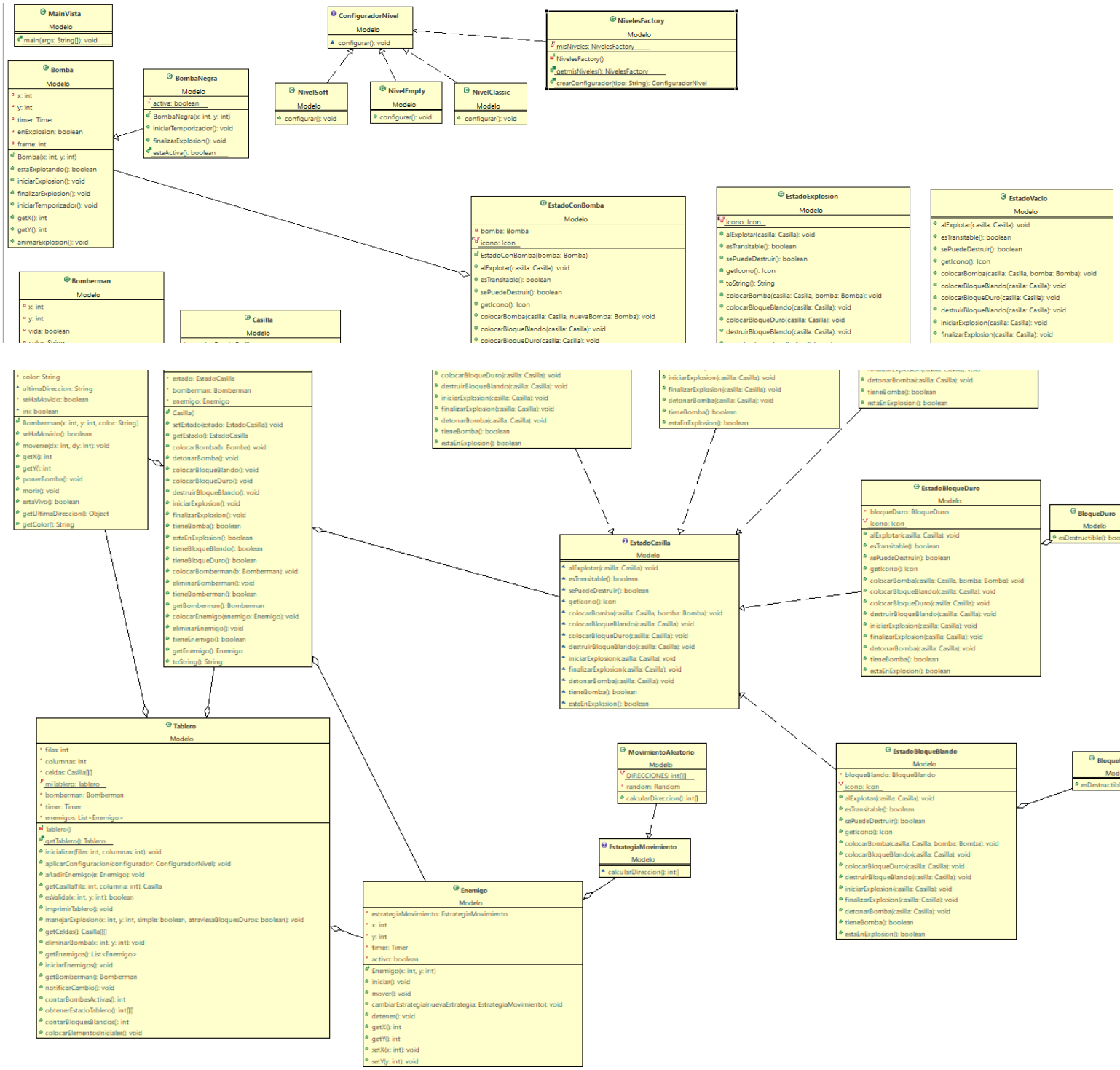
ÍNDICE

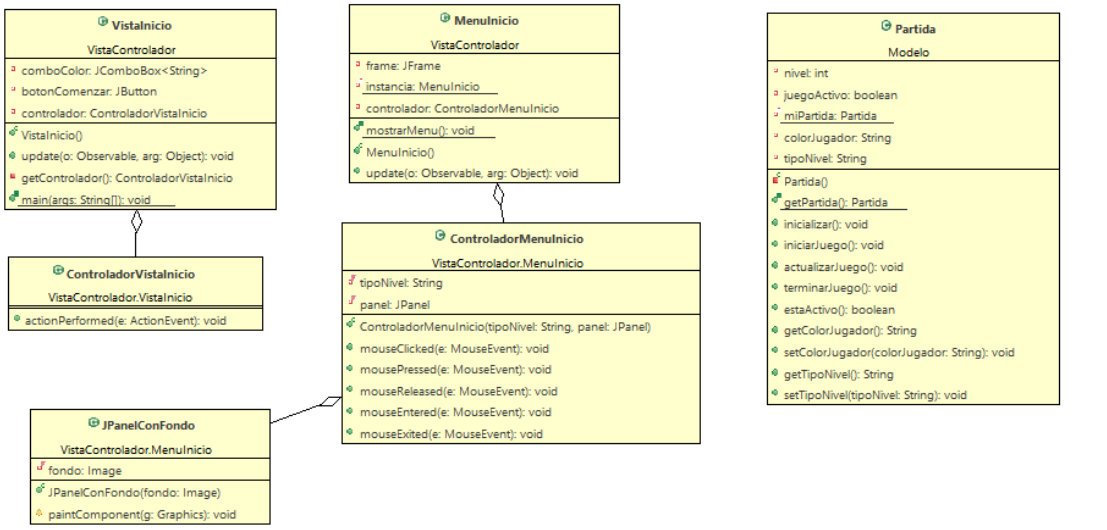
1. Introducción

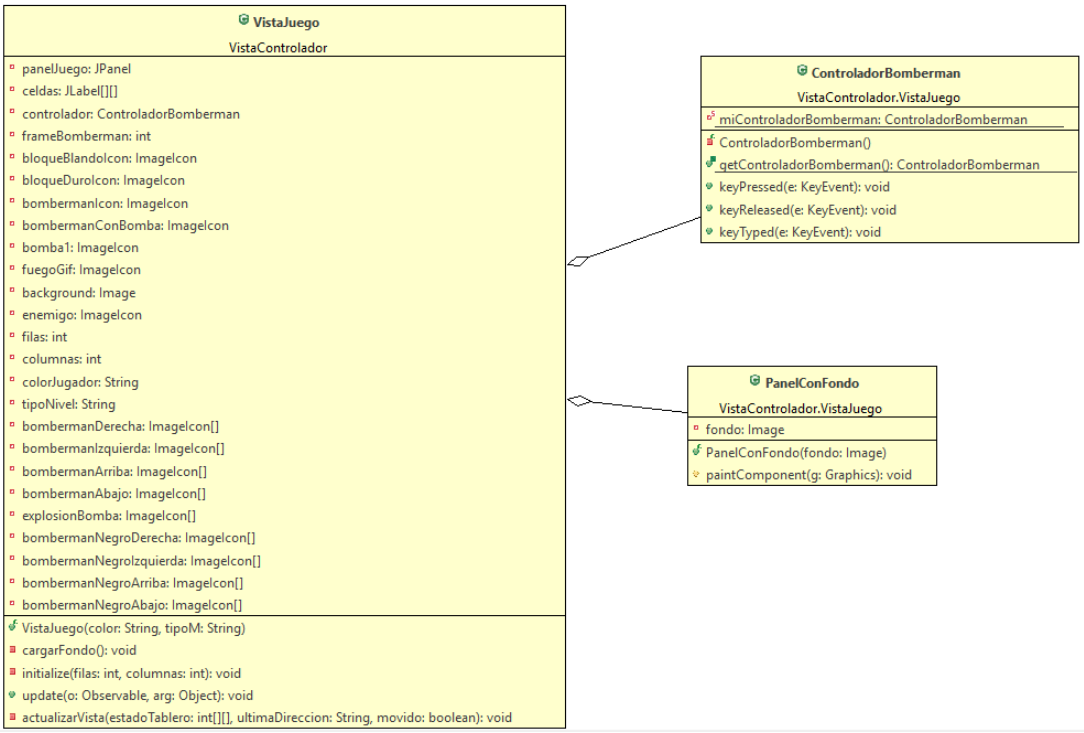
En este segundo sprint, hemos continuado con el desarrollo de nuestro proyecto partiendo de una versión funcional obtenida al finalizar el Sprint 1. El objetivo principal de esta fase ha sido mejorar la experiencia de juego. Entre las principales novedades, destacamos la inclusión de enemigos que aportan un nuevo nivel de desafío, un menú de inicio que permite personalizar aspectos del juego antes de comenzar, y la posibilidad de elegir un Bomberman con un color distinto, lo que mejora la identificación del personaje en pantalla. Además, se han realizado pequeños arreglos y mejoras internas para optimizar el funcionamiento general del juego.

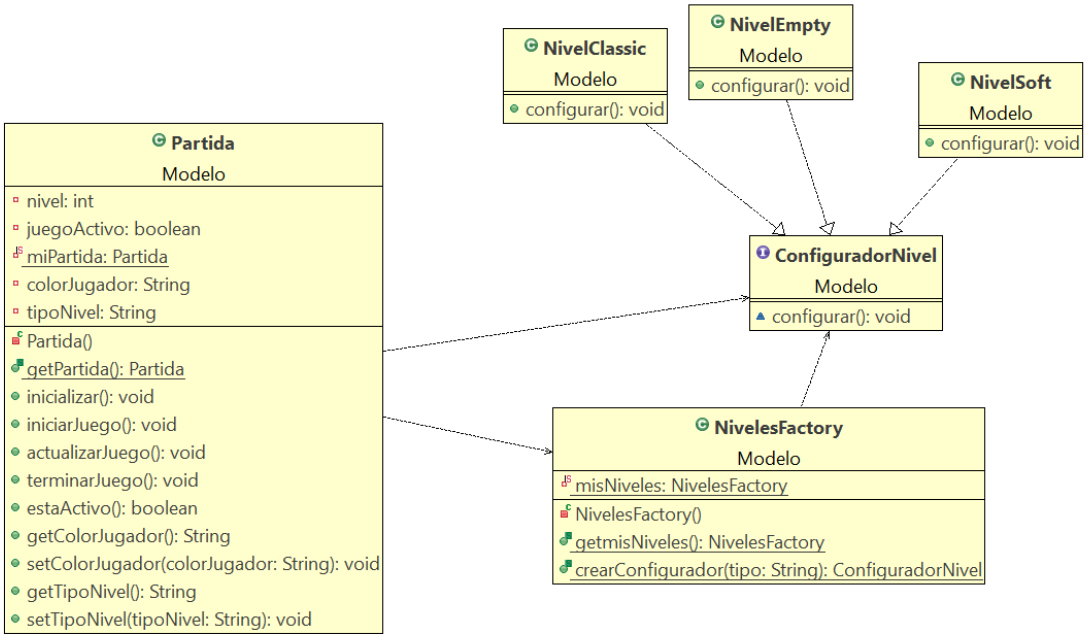
Imagen de la pantalla de inicio, selección de nivel y juego

2. Diagrama de clases




VistaInicio
VistaControlador
comboColor: JComboBox<String>
botonComenzar: JButton
controlador: ControladorVistaInicio
VistaInicio()
update(o: Observable, arg: Object): void
getControlador(): ControladorVistaInicio
main(args: String[]): void
MenuInicio
VistaControlador
frame: JFrame
instancia: MenuInicio
controlador: ControladorMenuInicio
mostrarMenu(): void
MenuInicio()
update(o: Observable, arg: Object): void
Partida
Modelo
nivel: int
juegoActivo: boolean
miPartida: Partida
colorJugador: String
tipoNivel: String
Partida()
getPartida(): Partida
inicializar(): void
iniciarJuego(): void
actualizarJuego(): void
terminarJuego(): void
estaActivo(): boolean
getColorJugador(): String
setColorJugador(colorJugador: String): void
getTipoNivel(): String
setTipoNivel(tipoNivel: String): void
ControladorVistaInicio
VistaControlador.VistaInicio
actionPerformed(e: ActionEvent): void
ControladorMenuInicio
VistaControlador.MenuInicio
tipoNivel: String
panel: JPanel
ControladorMenuInicio(tipoNivel: String, panel: JPanel)
mouseClicked(e: MouseEvent): void
mousePressed(e: MouseEvent): void
mouseReleased(e: MouseEvent): void
mouseEntered(e: MouseEvent): void
mouseExited(e: MouseEvent): void
JPanelConFondo
VistaControlador.MenuInicio
fondo: Image
JPanelConFondo(fondo: Image)
paintComponent(g: Graphics): void

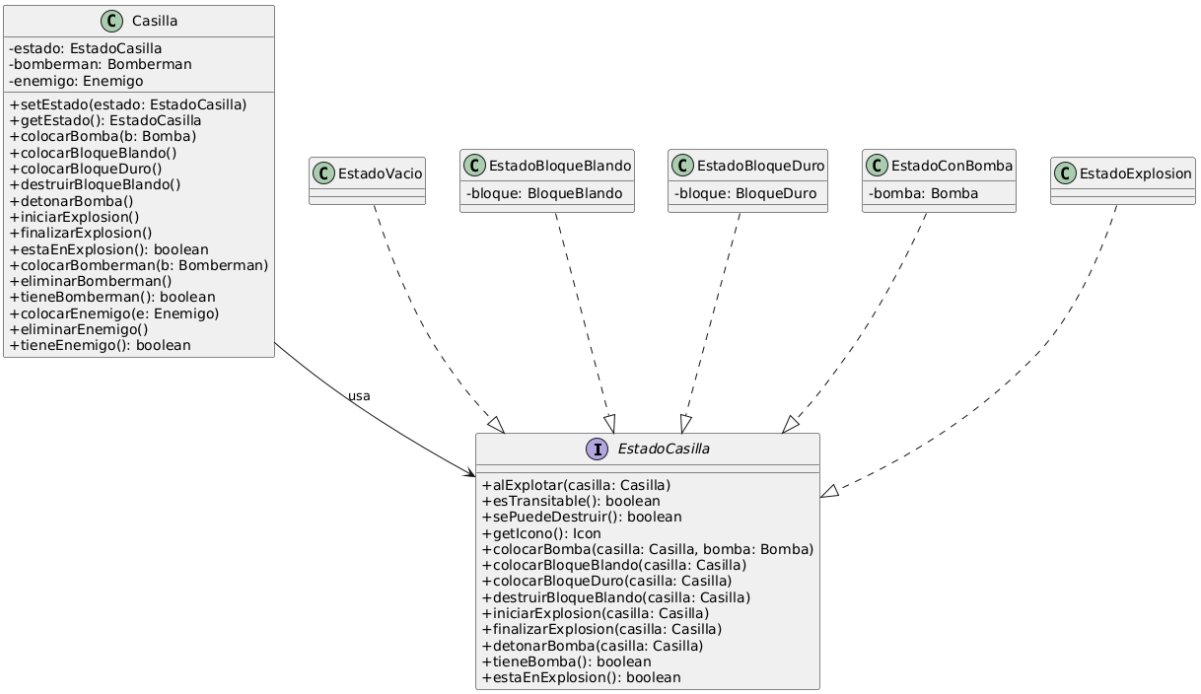

VistaJuego
VistaControlador
panelJuego: JPanel
celdas: JLabel[][]
controlador: ControladorBomberman
frameBomberman: int
bloqueBlandoIcon: ImageIcon
bloqueDuroIcon: ImageIcon
bombermanIcon: ImageIcon
bombermanConBomba: ImageIcon
bomba1: ImageIcon
fuegoGif: ImageIcon
background: Image
enemigo: ImageIcon
filas: int
columnas: int
colorJugador: String
tipoNivel: String
bombermanDerecha: ImageIcon[]
bombermanIzquierda: ImageIcon[]
bombermanArriba: ImageIcon[]
bombermanAbajo: ImageIcon[]
explosionBomba: ImageIcon[]
bombermanNegroDerecha: ImageIcon[]
bombermanNegroIzquierda: ImageIcon[]
bombermanNegroArriba: ImageIcon[]
bombermanNegroAbajo: ImageIcon[]
VistaJuego(color: String, tipoM: String)
cargarFondo(): void
initialize(filas: int, columnas: int): void
update(o: Observable, arg: Object): void
actualizarVista(estadoTablero: int[][], ultimaDireccion: String, movido: boolean): void
ControladorBomberman
VistaControlador.VistaJuego
miControladorBomberman: ControladorBomberman
ControladorBomberman()
getControladorBomberman(): ControladorBomberman
keyPressed(e: KeyEvent): void
keyReleased(e: KeyEvent): void
keyTyped(e: KeyEvent): void
PanelConFondo
VistaControlador.VistaJuego
fondo: Image
PanelConFondo(fondo: Image)
paintComponent(g: Graphics): void


NivelClassic
Modelo
configurar(): void
NivelEmpty
Modelo
configurar(): void
NivelSoft
Modelo
configurar(): void
Partida
Modelo
nivel: int
juegoActivo: boolean
miPartida: Partida
colorJugador: String
tipoNivel: String
Partida()
getPartida(): Partida
inicializar(): void
iniciarJuego(): void
actualizarJuego(): void
terminarJuego(): void
estaActivo(): boolean
getColorJugador(): String
setColorJugador(colorJugador: String): void
getTipoNivel(): String
setTipoNivel(tipoNivel: String): void
ConfiguradorNivel
Modelo
configurar(): void
NivelesFactory
Modelo
misNiveles: NivelesFactory
NivelesFactory()
getmisNiveles(): NivelesFactory
crearConfigurador(tipo: String): ConfiguradorNivel

3. Patrones

3.1. Patrón State para clase Casilla:

En este proyecto, el patrón State se ha aplicado a la clase Casilla, que representa cada celda del tablero del juego Bomberman. Dado que una casilla sólo puede estar en **un único estado a la vez** (vacía, con bomba, con bloque blando, en explosión, etc.), es necesario aplicar el patrón State. Cada estado representa un comportamiento distinto que puede cambiar dinámicamente durante la partida.



Casilla:
Mantiene una referencia al objeto EstadoCasilla actual y delega en él todas las acciones relacionadas con el comportamiento del terreno (poner bomba, explotar, etc.). Cambia su estado dinámicamente usando setEstado().

EstadoCasilla:
Define el conjunto de métodos que todos los estados concretos deben implementar (por ejemplo: detonarBomba(), iniciarExplosion(), esTransitable(), etc.).

EstadoVacio:

Representa una casilla vacía, transitable. Permite colocar bombas o bloques. Al recibir una explosión, pasa a EstadoExplosion, etc.

```
@Override
public void colocarBomba(Casilla casilla, Bomba bomba) {
    casilla.setEstado(new EstadoConBomba(bomba));
}

@Override
public void colocarBloqueBlando(Casilla casilla) {
    casilla.setEstado(new EstadoBloqueBlando());
}

@Override
public void colocarBloqueDuro(Casilla casilla) {
    casilla.setEstado(new EstadoBloqueDuro());
}
```

EstadoConBomba:
Representa una casilla que contiene una bomba. Gestiona la detonación y animación de la explosión. Al terminar la animación, transiciona a EstadoExplosion.

```
public void detonarBomba(Casilla casilla) {
    System.out.println("💣 Bomba detonada en casilla");

    EstadoExplosion explosion = new EstadoExplosion();
    casilla.setEstado(explosion);
    explosion.iniciarExplosion(casilla);

    bomba.animarExplosion();
}
```

EstadoBloqueBlando
Representa una casilla con un bloque destructible. Puede ser destruido por una explosión y pasaría a EstadoVacio.

```
@Override
public void destruirBloqueBlando(Casilla casilla) {
    casilla.setEstado(new EstadoVacio());
}
```

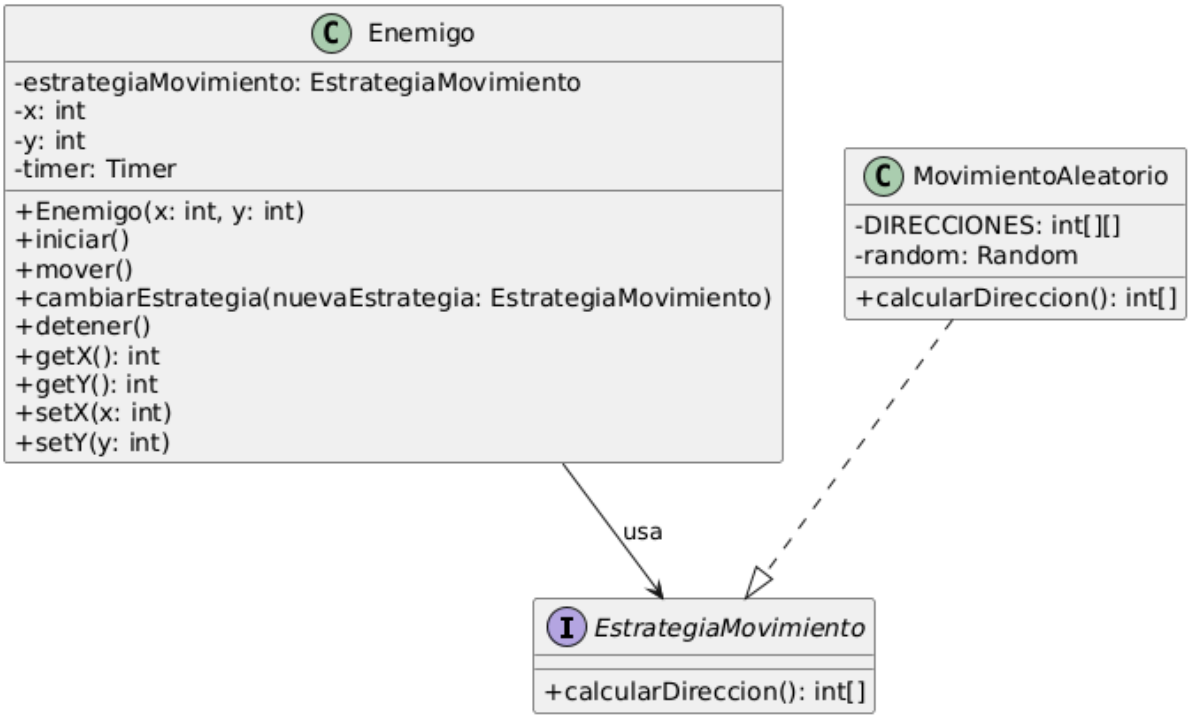
EstadoBloqueDuro:
Representa una casilla con un bloque indestructible. Ignora las explosiones. Contiene un BloqueDuro.

EstadoExplosion:
Representa una casilla en llamas (en proceso de explosión). Se convierte automáticamente en EstadoVacio tras un temporizador de 2 segundos.

```java
@Override
public void iniciarExplosion(Casilla casilla) {
    Timer timer = new Timer();
    timer.schedule(new TimerTask() {
        public void run() {
            casilla.setEstado(new EstadoVacio());
            Tablero.getTablero().notificarCambio();
        }
    }, 2000);

}
```

3.2. Patrón Strategy para el movimiento:

En este proyecto, el patrón Strategy se ha aplicado al movimiento del Enemigo. La idea es que tenga diferentes formas de moverse, pero todos comparten una interfaz para que el juego lo vea como movimiento.




```java
public interface EstrategiaMovimiento {
    int[] calcularDireccion();
}
```

Esta es la estrategia. Define el comportamiento que se espera: calcular una dirección de movimiento. Cualquier clase que implemente esta interfaz puede tener una lógica diferente para moverse.

```java
public class MovimientoAleatorio implements EstrategiaMovimiento {
    private static final int[][] DIRECCIONES = {
        {1, 0}, {-1, 0}, {0, 1}, {0, -1}, {0, 0}
    };

    private Random random = new Random();

    @Override
    public int[] calcularDireccion() {
        int accion = random.nextInt(5);
        return DIRECCIONES[accion];
    }
}
```

Este es un ejemplo de estrategia concreta: define un algoritmo de movimiento aleatorio. Cada vez que se llama a calcularDireccion(), devuelve una dirección al azar.
En el siguiente sprint implementaremos más estrategias concretas de movimiento.

3.3. Patrón Observer:

En este proyecto, el patrón Observer se ha aplicado para conectar de forma correcta el modelo con las vistas, a través de Partida y Tablero.

3.3.1 Tablero

Tablero es el observable que representa el propio juego, el tablero, al moverse un jugador o lanzar una bomba, el tablero notifica al cambio.

```java
public void notificarCambio() {
    setChanged();
    notifyObservers(new Object[]{bomberman.getUltimaDireccion(), obtenerEstadoTablero()});
}
```

3.3.2 Partida

En este sprint, hemos añadido partida como observable.Representa el estado general de la partida, cuando se hace algún cambio en el menú inicio o en VistaInicio se llama a setColorJugador y a setTipoNivel que notifican los cambios de la configuración del menú y permite que las vistas esten escuchando a Partida y se enteren del cambio y se actualicen.

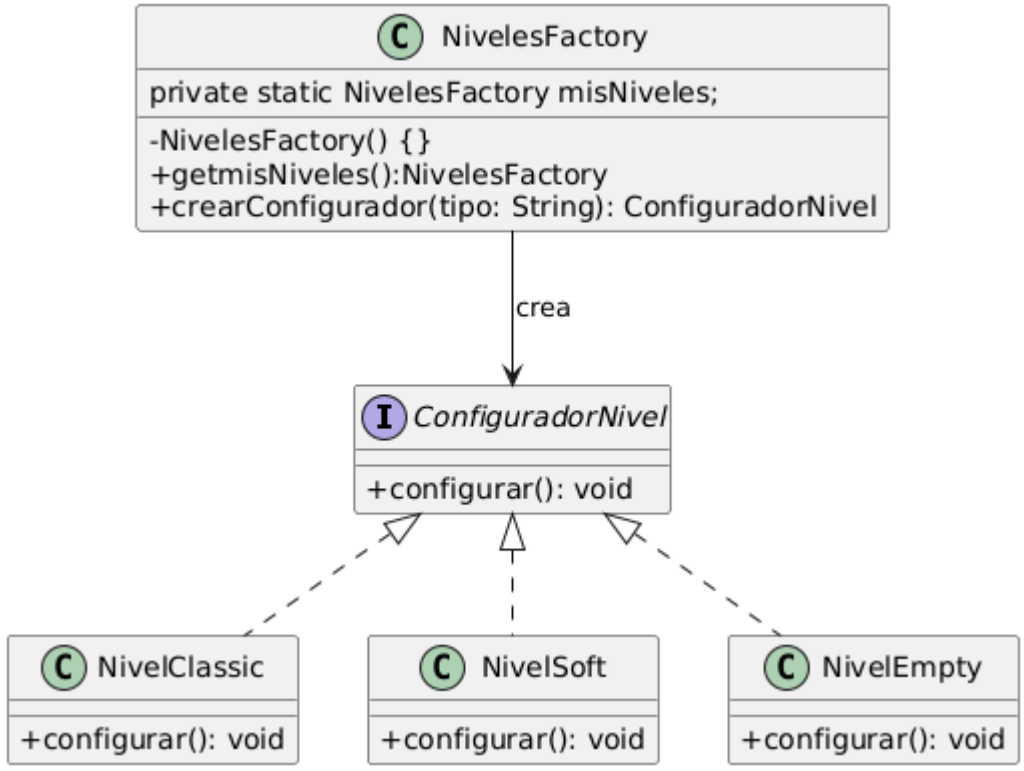
```java
public void setColorJugador(String colorJugador) {
    this.colorJugador = colorJugador;
    setChanged();
    notifyObservers(colorJugador);
}

public String getTipoNivel() {
    return tipoNivel;
}

public void setTipoNivel(String tipoNivel) {
    this.tipoNivel = tipoNivel;
    setChanged();
    notifyObservers(new Object[] { colorJugador, tipoNivel });
}
```

3.4. Patrón Factory para los niveles:

En esta parte del proyecto aplicamos el patrón Factory para encargarnos de configurar el tablero según el tipo de nivel elegido (por ejemplo: vacío, con bloques blandos, clásico…). Lo hicimos así para separar la lógica de creación del tablero del resto del código, y además

permitir que sea muy fácil añadir nuevos tipos de niveles en el futuro sin tener que modificar el núcleo del juego.

Se ha usado como interfaz porque cada tipo de nivel tiene su propia lógica completamente distinta, y no había código común que reutilizar. Si más adelante se viera que muchos niveles repiten cosas (como colocar enemigos o bordes), podríamos pasar a una clase abstracta con métodos compartidos.



ConfiguradorNivel:

Creamos una interfaz llamada **ConfiguradorNivel**, que define un único método:

```
3  public interface ConfiguradorNivel {
4      void configurar();
5  }
```

NivelesFactory:

```
public static ConfiguradorNivel crearConfigurador(String tipo) {
    return switch (tipo.toUpperCase()) {
        case "CLASSIC" -> new NivelClassic();
        case "SOFT" -> new NivelSoft();
        case "EMPTY" -> new NivelEmpty();
        default -> throw new IllegalArgumentException("Tipo de nivel desconocido: " + tipo);
    };
```

Esta clase es la que se encarga de crear el tipo de nivel que queremos según una cadena de texto (por ejemplo: "CLASSIC", "SOFT" o "EMPTY"). Mediante esa cadena de texto, devuelve una instancia concreta de la clase que configura ese nivel.

Todas estas clases implementan la interfaz **ConfiguradorNivel**, por lo que el código que la recibe no necesita saber qué tipo exacto de objeto está usando.

3 tipos de niveles:

NivelClassic

Este nivel coloca bloques duros en posiciones fijas con un patrón tipo ajedrez, y también bloques blandos repartidos por el resto del mapa. Deja algunas zonas sin nada para que el jugador pueda moverse al inicio.

```
Casilla casilla = Tablero.getTablero().getCasilla(i, j);
if (casilla == null) continue;

if (i % 2 == 1 && j % 2 == 1) {
    casilla.colocarBloqueDuro();
} else if (rand.nextDouble() < 0.4) {
    casilla.colocarBloqueBlando();
}
```

NivelSoft

Coloca una gran cantidad de bloques blandos por todo el mapa, excepto en el punto de inicio del jugador. Está pensado para ser un nivel más fácil, en el que se puedan destruir la mayoría de obstáculos.

NivelEmpty

Este nivel deja el mapa completamente vacío, salvo por el borde de bloques duros. Es útil para pruebas o para empezar desde cero.

```
@Override
public void configurar() {
    Tablero.getTablero().inicializar(11, 17);

    // Nivel vacío: no se colocan bloques ni obstáculos

    Tablero.getTablero().colocarElementosIniciales();
}
```

4. Planificación y reparto de tareas

➢ Planificación:

Día de inicio del Sprint : 27 de marzo, grupo 2, miércoles 17.00-19.00

- **Semanas de 24 al 28 de marzo :**

- Revisión del primer sprint.
- Reunión de planificación: distribución de tareas y elección de patrones.
- Debate sobre el patrón State para las casillas, Factory para niveles y Strategy para enemigos.
- Inicio de bocetos para pantalla de inicio y menú.

- **Semanas de 31 al 6 de abril :**

- Implementación de enemigos
- Desarrollo del menú (con niveles)
- Patrón state casilla

- **Semana del 7 al 13 de marzo:**

- Bomberman Negro
- Patrón Factory niveles
- Integración de todos los módulos.
- Añadida lógica a EstadoExplosion con TimerTask.

- **Semana del 14 al 20 de marzo:**

- Corrección de bugs (fuegos que no desaparecen, estado de la bomba que se quedaba pillado, no morían los enemigos, se movían cuando se movía el bomberman).
- Ajustes gráficos, pruebas de niveles y revisión del patrón MVC.
- Revisión final del juego.

- Capturas de pantalla, generación de diagramas UML.
- Preparación del informe de patrones.
- Documentación de clases.

➢ Reparto de tareas

Miembros	Tareas Principales	Horas estimadas (laboratorios)	Horas reales	Comentarios
Paula	Diseño e implementación de la pantalla de inicio con opciones básicas. Integración con el menú	8h	8h 30m	Ajustes visuales añadieron trabajo.
Aitzol	Implementación de NivelesFactory y clases NivelClassic, NivelSoft, NivelEmpty. Menú funcional con elección de niveles.	7h	8h	Hubo cambios de estructura que requirieron refactor
Mario	Implementación de la clase Enemigo y estrategia de movimiento aleatorio (Strategy)	9h	10h 20m	Resolución de colisiones con el bomberman
Alvaro	Lógica y sprites del Bomberman Negro. Gestión de input.	7h	5h 30m	Sin problemas
Eneko	Aplicación del patrón State a Casilla, refactor de clases de estado, pruebas e integración	8h	9h	Depuración y pruebas por bugs visuales

El trabajo en grupo fue clave: aunque cada uno tenía una tarea principal, nos apoyamos mutuamente en todas.