CODIGO CERO
INFORME FINAL

Iker Fernández, Eneko Rodríguez, María Fernández,
Urko Horas y June Castro

PROFESOR:
Ander Barrena
CURSO: 2

ÍNDICE

1. PRESENTACIÓN Y OBJETIVOS

La realización de este proyecto ha consistido en crear un juego llamado Bomberman, el cual se ha empezado desde cero con la ayuda que se ha recibido en las horas de laboratorio de esta asignatura.

Primero de todo, se va a hacer una breve explicación sobre en qué consiste el juego. El objetivo principal de Bomberman es eliminar a los enemigos y sobrevivir en un laberinto lleno de bloques y obstáculos. El juego se desarrolla en un mapa donde puede haber bloques duros y bloques blandos, que dependiendo de que tipo sean pueden ser destruidos por bombas. El personaje, es decir, el bomberman puede colocar bombas que explotan en cruz (arriba, abajo, izquierda y derecha) después de unos segundos. Sobre el bomberman y las bombas, puede haber dos tipos: está el bomberman blanco, el cual dispone de una bomba que su rango de explosión es 1, mientras que el bomberman negro tiene una bomba cuyo rango de explosión es de las mismas medidas del tablero. A medida que se han ido completando tareas, se ha ido implementando más funciones para que el juego sea más completo y efectivo.

Todo el grupo ha colaborado para que se lleven a cabo todas las entregas dentro del límite establecido y, además, poder ir aprendiendo funciones que antes no se sabían. Por último, se han cumplido todos aquellos objetivos, los cuales se explicarán más adelante, que se habían puesto antes de comenzar este trabajo.

Objetivos:

- → El bomberman blanco pudiera moverse por el tablero
- → Las bombas explotaran tras varios segundos y solo rompieran los bloques blandos
- → Aplicar el patrón MVC de manera correcta
- → Crear nuevos mapas
- → Implementar las mismas funciones para el bomberman negro
- → Implementar enemigos que aparezcan de manera aleatoria
- → Aplicar los patrones necesarios de manera adecuada

Además de estos objetivos, se decidió implementar varias funciones opcionales:

- → Un final Boss
- → Power Up

→ Una opción en la pantalla donde muestra el tiempo que llevas jugando la partida, el número de enemigos que quedan por matar, el número de vidas que le quedan al bomberman y si ha escogido la opción del Power Up.

2. GESTIÓN Y REPARTO DE TAREAS

Para seguir con el desarrollo de este proyecto, se han tenido en cuenta las nuevas actualizaciones que se han pedido en este sprint. Para ello, todos los miembros del grupo han participado con el objetivo de desarrollar correctamente aquellas nuevas funcionalidades.

Miembros del grupo:

- **Componente 1:** Iker Fernández
- **Componente 2:** Eneko Rodríguez
- **Componente 3:** María Fernández
- **Componente 4:** Urko Horas
- **Componente 5:** June Castro

Reparto de tareas:

- **Creación del código:**
  - PowerUp: Urko y Eneko
  - Bomba Diagonal: Iker
  - Boss Final: Iker
  - Java 8: Iker
  - HUD (in game): María y June

- **Diseño Diagrama de Clases:** Iker Fernández

  Esta tarea ha conllevado 2 horas, ya que, se han actualizado y añadido, métodos y clases. Se ha realizado de manera manual con la herramienta draw.io.

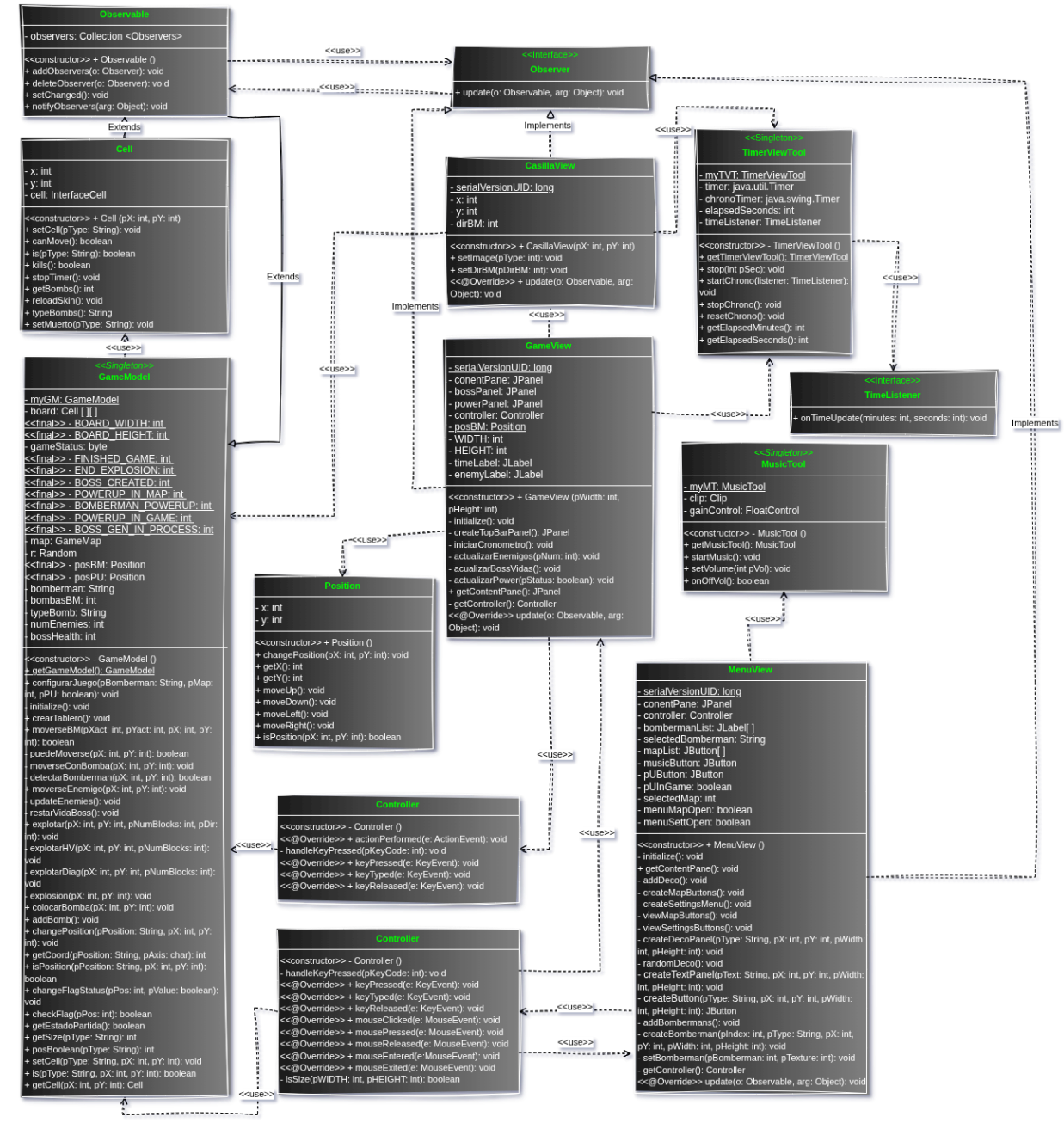
3. DIAGRAMA DE CLASES

3.1. MODELO

Pulsa **AQUÍ** para ver el diagrama en mayor tamaño

3.2. MVC

AQUÍ CONTINUARÍA EL DIAGRAMA DE CLASES DEL MODELO



Pulsa **AQUÍ** para ver el diagrama en mayor tamaño

4. INFORME PATRONES Y MVC

4.1. PATRÓN MVC

El patrón MVC (Modelo-Vista-Controlador) es un patrón de diseño, enfocado en organizar la lógica interna de una aplicación. El MVC organiza el código en los tres componentes de los que se compone su nombre, lo que hace que se facilite el mantenimiento y escalabilidad del código, además de permitir cambiar cualquiera de las capas sin que esta modificación afecte al resto.

1. **Modelo:**
   El modelo representa los datos y la lógica de negocio. Además, se encarga de recuperar y manipular los datos, como gestionar la información y reglas del juego en este caso.

   En nuestro proyecto, el modelo está dentro del paquete “model”, y este, a rasgos generales, se encarga de gestionar el estado del tablero (controlando cómo se genera el mapa, cómo se mueve el bomberman, cómo se dan las explosiones y cómo se finaliza la partida), las celdas (patrón observer), los bloques, los enemigos, las bombas y las explosiones, controlando la lógica de cada elemento del juego.

2. **Vista:**
   La vista representa la interfaz gráfica que se encarga de mostrarle la información al usuario. No contiene lógica de negocio, solamente la representación gráfica, y su única función es actualizarse cuando el modelo cambia.

   En nuestro proyecto, la vista está dentro del paquete “viewController”, que se encarga de la representación gráfica del juego. Hace su función en las clases “CellView” (representa las celdas del tablero, y se actualiza mediante el patrón observer) y “GameView” (un JFrame donde se visualiza el tablero).

Controlador:
El controlador se encarga de la gestión de la interacción entre el usuario, la vista y el modelo, es decir, recibe las entradas del usuario, llama a métodos del modelo para modificar los datos y hace que la vista se actualice.

En nuestro proyecto, el controlador está en una clase privada llamada "Controller" dentro de la clase "GameView". En este caso se encarga de detectar cuando el usuario pulsa una tecla, y según detecta la acción, llama al modelo (Tablero) que realizará los cambios correspondientes.

4.2. PATRONES

Los patrones de diseño son soluciones reutilizables que sirven para ayudar a resolver alguno de los problemas surgidos en el desarrollo de software. Son guías que ayudan a estructurar y organizar el código que se necesita para desarrollar un proyecto de manera eficiente y flexible. En este caso, se han implementado los siguientes cuatro patrones:

4.2.1. OBSERVER

El patrón Observer es un patrón de comportamiento que permite que un objeto notifique automáticamente a diferentes observadores cuando su estado cambia, manteniendo un bajo acoplamiento entre ellos.

Es muy útil porque facilita la comunicación entre objetos sin que dependan directamente entre sí y permite que múltiples observadores reaccionen a cambios en el sujeto. Para añadir, reduce el acoplamiento entre componentes.

Está implementado a través de las clases Cell y GameModel, que heredan de Observable, y de CellView, GameView y MenuView que actúan como Observer. Cuando el estado de una celda cambia (por ejemplo, tras una explosión o movimiento), esta notifica automáticamente a su vista asociada para que se actualice. Esto permite una sincronización eficiente entre el modelo y la interfaz gráfica.

4.2.2. STATE

El patrón State es un patrón de comportamiento que permite que un objeto modifique su comportamiento cuando cambia su estado interno, simulando que cambia su clase. Es útil cuando un objeto debe cambiar su comportamiento dependiendo de su estado actual, manteniendo el código más limpio, organizado y fácil de extender.

Entre sus ventajas se encuentran, la separación de la lógica de los estados en clases individuales y, además, evita el uso de múltiples estructuras if-else o switch-case. Por lo tanto, permite agregar nuevos estados sin modificar la clase principal.

Se utiliza en las celdas del tablero. Cada instancia de Cell contiene un objeto que representa su tipo (bomba, bloque, vacío, enemigo...), y este objeto define su propio comportamiento. Así, el comportamiento de la celda depende de su estado interno, permitiendo una estructura más clara y extensible para manejar distintos tipos de elementos del juego.

4.2.3. FACTORY

El patrón Factory es un patrón generativo que define una interfaz o clase abstracta para la creación de objetos, permitiendo a las demás subclases decidir qué clase concreta se debe instanciar.

Además, centraliza la creación de objetos y permite la creación de diferentes tipos de objetos sin modificar el código cliente. Puede facilitar la escalabilidad y el mantenimiento del código.

Está presente en clases como BlockFactory, que se encarga de crear bloques del juego según el tipo indicado. Gracias a este patrón, se separa la lógica de creación del resto del modelo, lo que permite instanciar nuevos elementos sin modificar el código cliente.

4.2.4. STRATEGY

El patrón Strategy también es un patrón de comportamiento que permite definir diferentes algoritmos dentro de una familia y se pueden seleccionarlos dinámicamente en tiempo de ejecución, sin ninguna modificación del código del cliente.

Algunas de las ventajas de utilizar este patrón es que separa la lógica de los algoritmos en clases independientes y facilita la sustitución y ampliación de estrategias sin modificar el código que ya existe. Además, reduce el acoplamiento y mejora la mantenibilidad.

En este proyecto, ha sido utilizado para gestionar la generación dinámica de mapas dentro del juego. Se define una interfaz llamada GameMap, que es implementada por varias clases concretas (GameMap1, GameMap2, GameMap3), cada una con su propia estrategia para generar el tablero correspondiente.

5. JAVA 8

Uno de los puntos a implementar en el proyecto era Java 8. Aunque ha sido un reto encontrar casos en los que se pudiese utilizar esta herramienta, se han encontrado algunos casos en los que resultarían incluso en una mejoría del código. Estos casos van a ser explicados a continuación.

5.1. CONTEO DE ENEMIGOS

Diferenciamos dos momentos en los que los enemigos del juego tienen que ser contados. Por un lado, al principio, tras aleatoriamente haber generado el mapa, con un número indefinido de enemigos, aunque siempre con un total menor o igual a 10. Por otro lado, según se van utilizando las diferentes habilidades que pueden eliminar a los enemigos. En este caso, se trata con la matriz de int que se rellena con la estrategia de GameMap correspondiente.

En ambos casos, se ha utilizado el mismo método:

```
(int)Arrays.stream(tab).flatMapToInt(Arrays::stream).filter(num->num==3).count();
```

En el que:

- Arrays.*stream*(tab), se refiere a la creación de streams para arrays.
- flatMapToInt(Arrays::*stream*), convierte la matriz de 2 dimensiones, a un stream unidimensional, para así poder tratarlo.
- filter(num->num==3), el cual filtra todos los Integer que sean 3, el cual corresponde al id de los enemigos.
- .count(), que cuenta el número de elementos restantes tras el anterior filtrado.

El método count() devuelve un long, como tenemos certeza de que no va a ser un número grande, hacemos un casting a int. Este valor será guardado en la variable numEnemies, que será mostrado gráficamente y marcará la aparición del Boss.

5.2. DETECCIÓN DE LA POSICIÓN DEL BOSS

Tras la muerte del Boss, ya que, a diferencia del Bomberman, de este no guardamos sus coordenadas, necesitamos encontrar su posición. De esta manera, su casilla notificará a su CasillaView correspondiente para que cambie la imagen a la del Boss muerto. En este ejemplo, hacemos la búsqueda en la matriz de casillas, por lo que, se trata con board.

```
Arrays.stream(board).flatMap(Arrays::stream).filter(cell-> cell.is("Boss")).forEach(cell->cell.setMuerto("Boss"));
```

En el que:

- Arrays.*stream*(board), se refiere a la creación de streams para arrays.
- flatMap(Arrays::*stream*), convierte la matriz de 2 dimensiones, a un stream de celdas unidimensional, para así poder tratarlo.
- filter(cell-> cell.is("Boss")), el cual filtra todas las celdas que sean de tipo Boss (Se sabe que sólo hay una, pero a futuro facilitaría el trabajo de cara a encontrar todos los Bosses)
- forEach(cell->cell.setMuerto("Boss")), se llama al método setMuerto("Boss"), el cuál notifica a su CasillaView para que cambie la imagen, para todas las celdas restantes tras el filtrado.

6. DESARROLLO

Este apartado del informe explicará cómo ha sido el desarrollo del proyecto durante los 3 sprints trabajados, comentando los objetivos que se han llevado a cabo en cada uno de ellos.

Durante el primer sprint, nos organizamos para desarrollar la base del proyecto. Nos llevó alrededor de 3 horas, y nos reunimos con el fin de poder decidir la mejor manera de llevar a cabo el proyecto. A partir de aquí, hicimos el reparto de tareas para que el trabajo estuviera dividido, aunque en todo momento estuvimos disponibles para ayudar a otros compañeros en su trabajo. Para ello, dividimos el grupo en 3 subgrupos que mantuvimos durante el resto del trabajo. Este modo de trabajo ha hecho la organización menos tediosa. Dividir las tareas en tareas más pequeñas simplifica mucho el trabajo y permite implementar las funcionalidades poco a poco.

En este sprint, construimos la base de lo que sería todo el trabajo, que consistía, entre otras cosas, en implementar el movimiento del bomberman o la colocación de las bombas, así como, la explosión de estas. Con el fin de interconectar el modelo con el view, implementamos el patrón MVC, el cual fue visto en clase.

Para el segundo sprint, implementamos el nuevo bomberman, el negro, con sus funcionalidades y su nueva bomba. Además, añadimos los enemigos y su movimiento e implementamos que todos los elementos funcionasen en conjunto con las interacciones que deben realizar. Para estas adiciones, fueron muy útiles los patrones trabajados en clase, ya que simplificaban la implementación de ciertas partes. Patrones como el Factory o el State han sido extremadamente útiles en el desarrollo del proyecto.

Para el tercer y último sprint, decidimos que cada uno de los subgrupos pensara alguna nueva función a implementar. Este sprint ha sido el menos costoso de los 3, ya que, cada pequeño grupo podía elegir lo que implementar y era más sencillo desarrollar algo pensando exactamente cómo se haría durante el proceso de tomar la decisión de lo que añadir.

Además, se ha implementado Java8 para contabilizar el número de enemigos en el mapa y, además, para encontrar la posición del boss final, que aparece una vez se han eliminado a todos los enemigos.

7. REPARTO DE TAREAS

7.1. SPRINT 1

TAREA	RESPONSABLES	TIEMPO PLANIFICADO	TIEMPO REAL
Bomba	June Castro María Fernández	2h	2h
Bomberman	Eneko Rodríguez Urko Horas	3h	4h
Explosión	June Castro María Fernández	2h	2h
Bloques	Iker Fernández	2h	4h
Tablero	Iker Fernández	4h	4h
ViewPantalla	Todos	3h	1h
ViewCasilla	Iker Fernández	3h	3h
InterfaceCell	Todos	2h	3h

7.2. SPRINT 2

TAREA	RESPONSABLES	TIEMPO PLANIFICADO	TIEMPO REAL
Bomberman	Eneko Rodríguez Urko Horas	1h	2h
Bombas	Eneko Rodríguez Urko Horas	3h	4h
Enemigos	María Fernández June Castro Iker Fernández	4h	6h
Mapas	Iker Fernández	1h	3h
Menú y utilidades	Iker Fernández	1h	3h

7.3. SPRINT 3

TAREA	RESPONSABLES	TIEMPO PLANIFICADO	TIEMPO REAL
PowerUp	Eneko Rodríguez Urko Horas	2h	3h
Bomba diagonal	Iker Fernández	1h	2h
Boss Final	Iker Fernández	4h	2h
Java 8	Iker Fernández	1h	1h
HUD (in game)	June Castro María Fernández	3h	3h

8. CONCLUSIONES

Este proyecto ha representado una valiosa experiencia de aprendizaje, ya que, nos ha brindado la oportunidad de aplicar en un contexto real todos los conocimientos adquiridos en clase.

Con el fin de cumplir con los requisitos establecidos, hicimos uso de varios patrones de diseño. Por un lado, trabajar con el patrón MVC nos permitió comprender la importancia de separar responsabilidades dentro de una aplicación, lo cual facilitó tanto el desarrollo como el mantenimiento del código. Por otro lado, integrar patrones adicionales como Strategy y Factory contribuyó a que nuestro código fuera más reutilizable, flexible y fácil de modificar. Esto nos permite implementar cambios sin tener que reestructurar grandes partes del sistema.

Algo que debemos destacar, es que a pesar de que todos debíamos realizar el mismo videojuego, había libertad en cuanto al diseño y el modo de juego. Esto fue una gran motivación, ya que, nos dio pie a la creatividad y tuvimos la oportunidad de darle nuestro toque personal a este proyecto.

Aunque el resultado final del proyecto ha sido satisfactorio y se ajusta a nuestras expectativas, el camino no fue del todo sencillo. Nos enfrentamos a varias dificultades, como, por ejemplo, diseñar la lógica para manejar el movimiento coordinado de todos los enemigos. Este desafío nos exigió muchas horas de trabajo y reflexión, pero creemos que ese esfuerzo se refleja claramente en la calidad del resultado obtenido.

9. GITHUB

El enlace del repositorio de GitHub, donde se encuentra el código, es el siguiente:

https://github.com/ikerfernandezmolano/CODIGO_CERO