# CODIGO CERO
# INFORME SPRINT 2

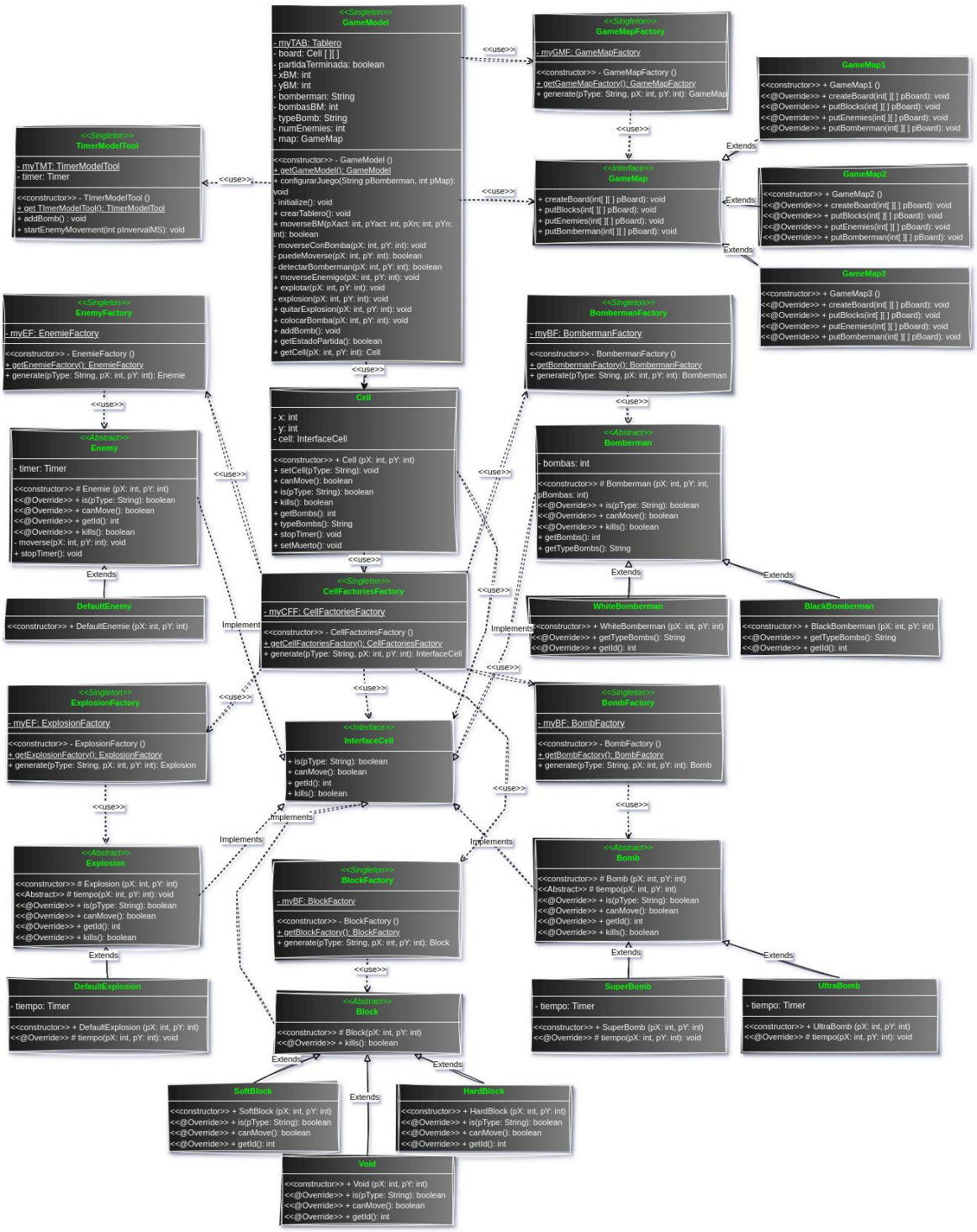
Iker Fernández, Eneko Rodríguez, María Fernández,
Urko Horas y June Castro

**PROFESOR:**
Ander Barrena
**CURSO:** 2

# ÍNDICE

# 1. DIAGRAMA DE CLASES

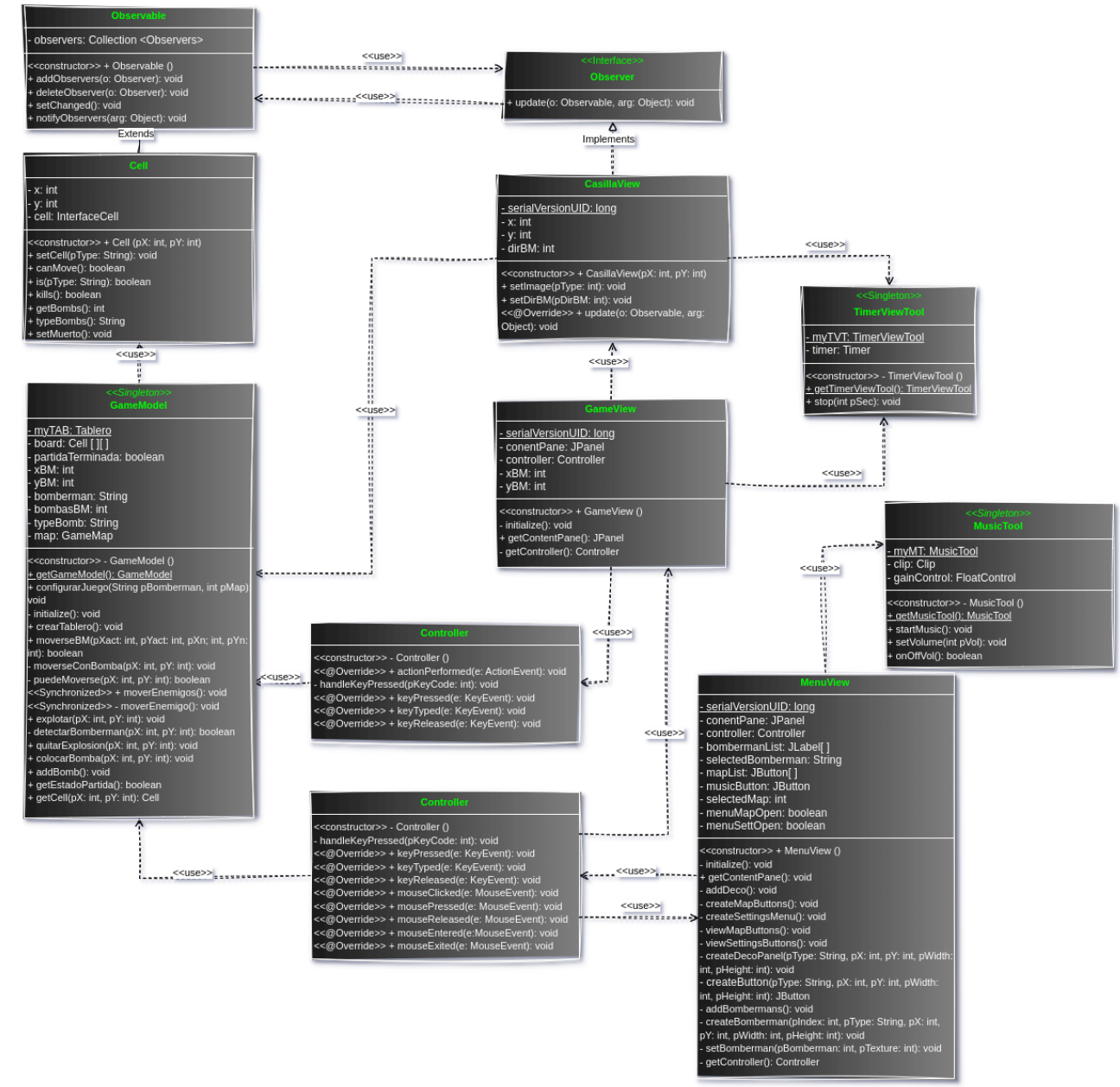
## 1.1. MODELO




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

# 1.2. MVC

AQUÍ CONTINUARÍA EL DIAGRAMA DE CLASES DEL MODELO




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

# 2. PLANIFICACIÓN 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:**
  - Bomberman: Urko y Eneko
  - Bombas: Urko y Eneko
  - Enemigos: María, June e Iker
  - Mapas: Iker
  - Menú y utilidades: Iker

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

  Esta tarea ha conllevado entre 1 y 2 horas ya que se han tanto actualizado, como añadido, métodos y clases. Se ha realizado de manera manual con la herramienta draw.io.

- **Documento Informe Patrones:**

| Sprint | Tarea | Responsable | Tiempo Planificado | Tiempo Real | Comentarios |
|---|---|---|---|---|---|
| 2 | Creación de código | Todos | 10 horas | 18 horas | Han surgido varios problemas que no esperábamos |
| 2 | Diseño Diagrama de Clases | Iker | 1 hora | 1 hora y media | Se ha intentado ponerlo de manera clara y que se entienda |
| 2 | Reparto de Tareas | María | 30 mins | 30 mins | Se ha organizado de |

| | | | | | |
|---|---|---|---|---|---|
| | | | | | manera correcta y clara |
| 2 | Informe Patrones | Urko, Eneko, María y June | 1 hora | 2 horas y media | Se ha detallado cada patrón utilizado en este proyecto |

# 3. INFORME PATRONES

## 3.1. DESCRIPCIÓN GENERAL

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 tres patrones:

### 3.1.1. 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.

### 3.1.2. 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.

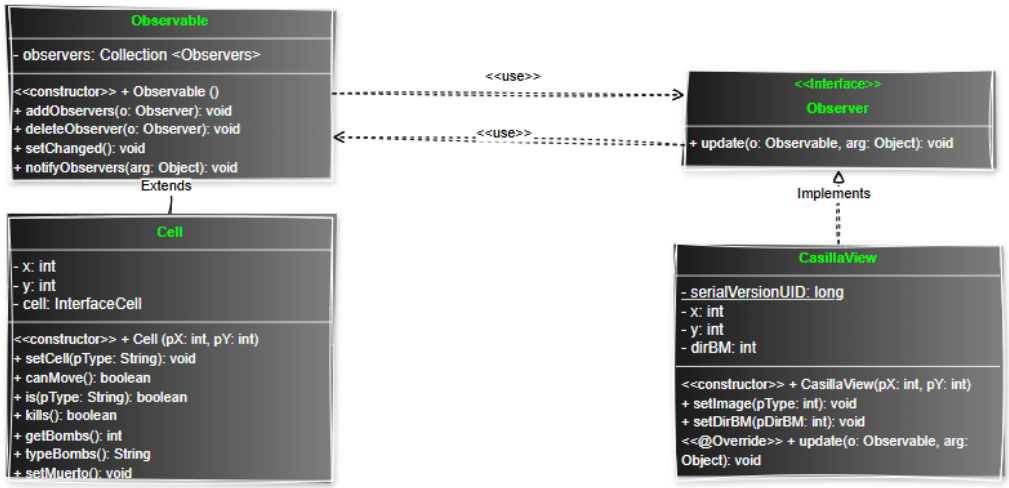
Algunas de las ventajas de utilizar este patrón es que separa la lógica de los algoritmos en clases independientes y facilita la sustitución y ampliación de estrategias sin modificar el código que ya existe. Además, reduce el acoplamiento y mejora la mantenibilidad.

### 3.1.3. 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.

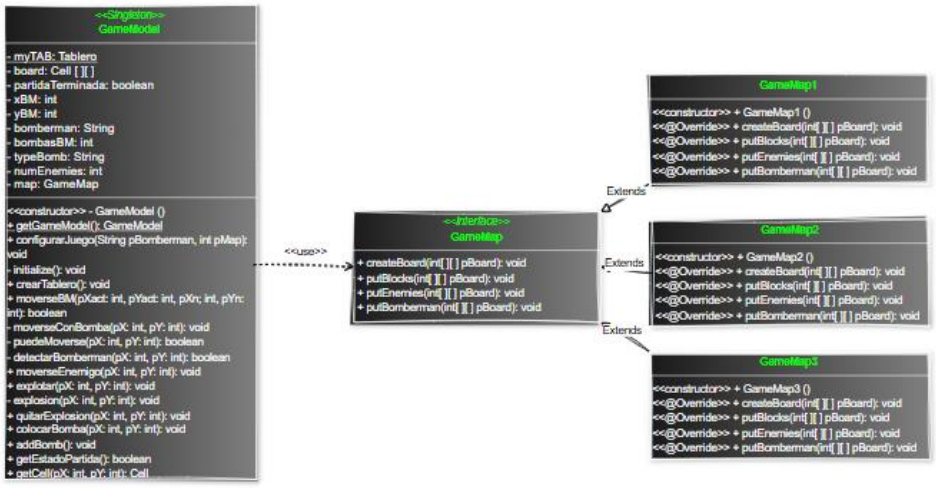
# 3.2. DIAGRAMAS DE LOS PATRONES

## 3.2.1. OBSERVER

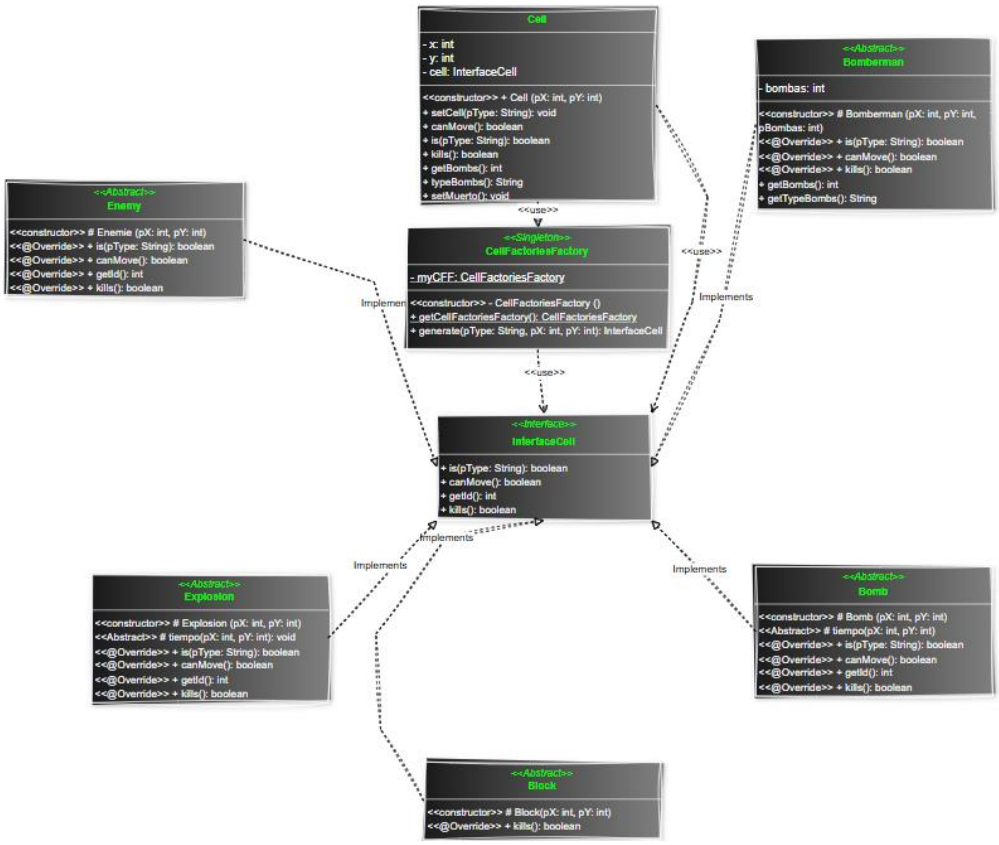



## 3.2.2. STRATEGY

### *3.2.2.1. GAMEMAP*

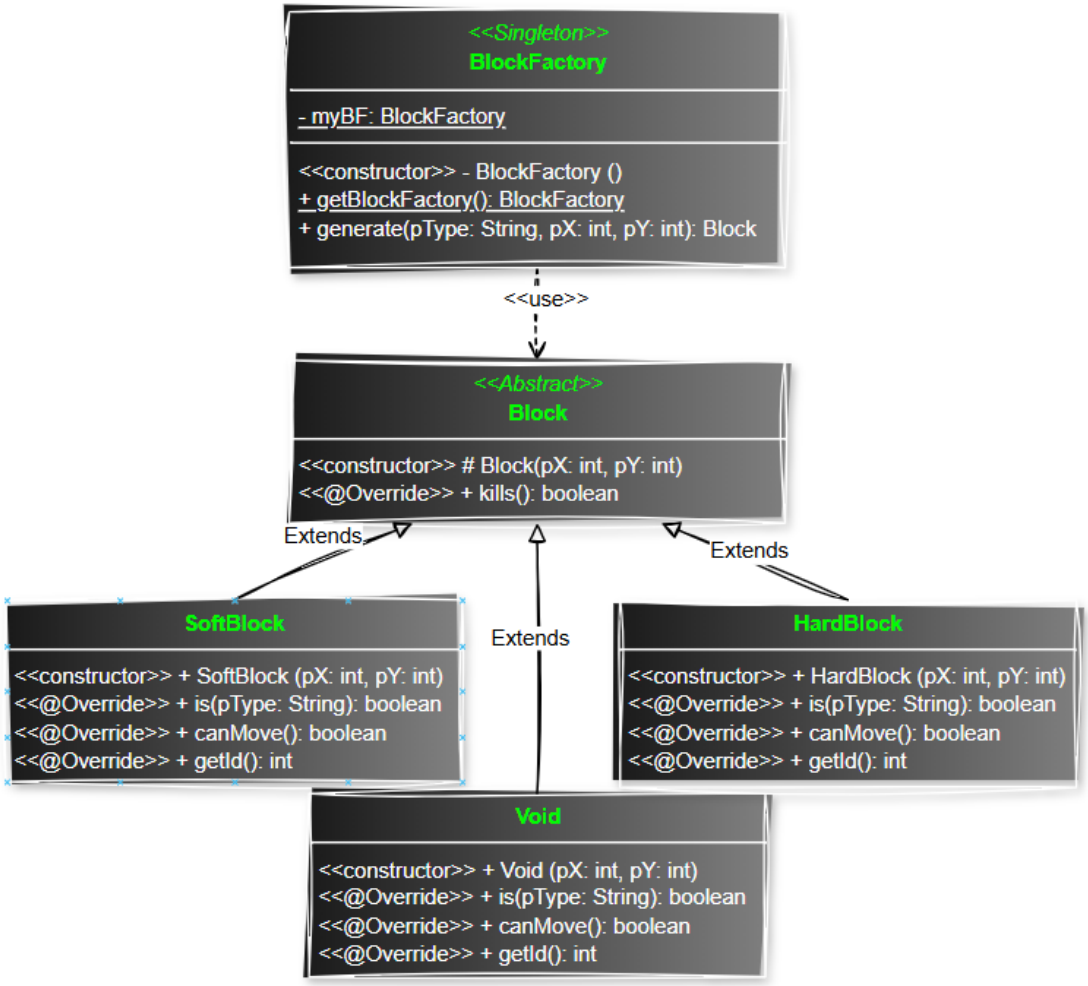



### *3.2.2.2. CELL*

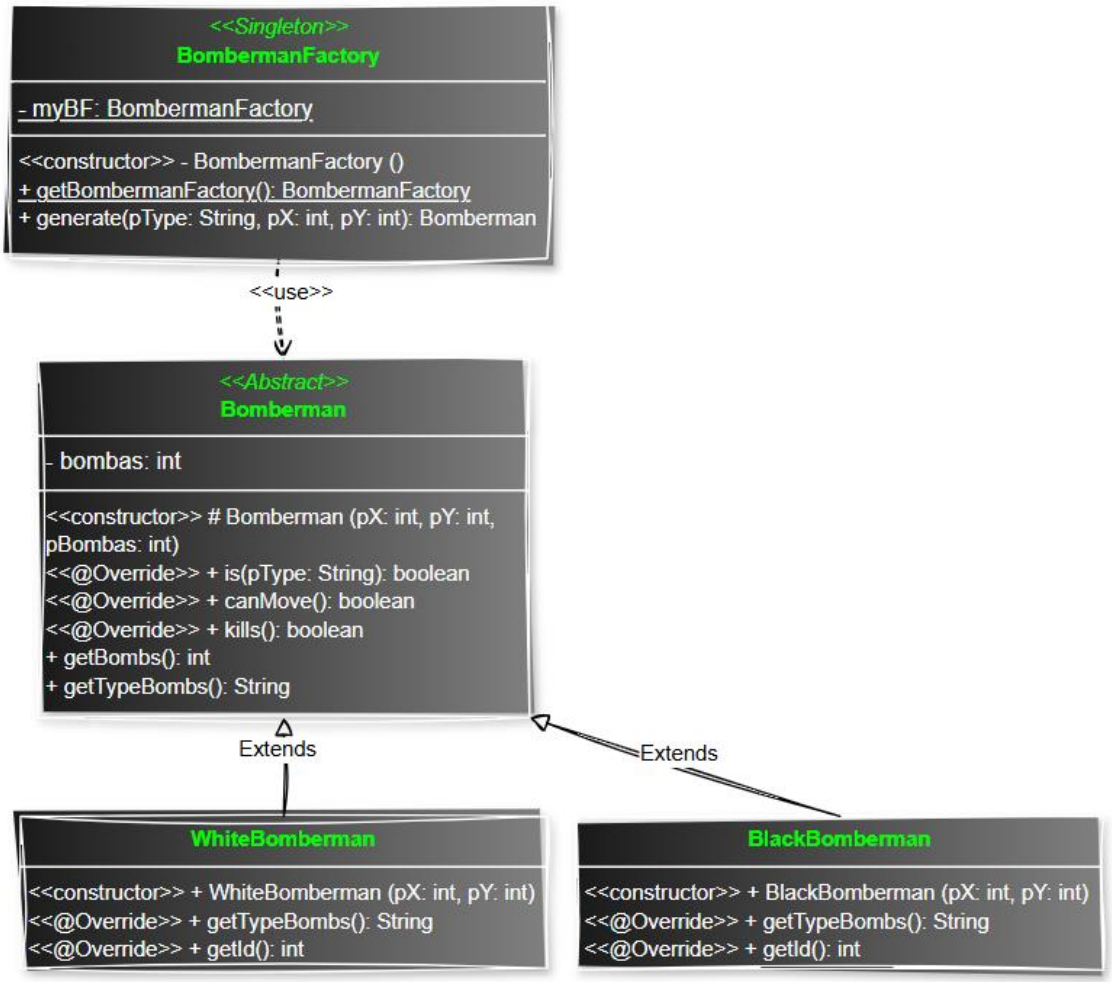

Cell
- x: int
- y: int
- cell: InterfaceCell
<<constructor>> + Cell (pX: int, pY: int)
+ setCell(pType: String): void
+ canMove(): boolean
+ is(pType: String): boolean
+ kills(): boolean
+ getBombs(): int
+ typeBombs(): String
+ setMuerto(): void
<<Abstract>>
Bomberman
- bombas: int
<<constructor>> # Bomberman (pX: int, pY: int, pBombas: int)
<<@Override>> + is(pType: String): boolean
<<@Override>> + canMove(): boolean
<<@Override>> + kills(): boolean
+ getBombs(): int
+ getTypeBombs(): String
<<Abstract>>
Enemy
<<constructor>> # Enemie (pX: int, pY: int)
<<@Override>> + is(pType: String): boolean
<<@Override>> + canMove(): boolean
<<@Override>> + getId(): int
<<@Override>> + kills(): boolean
<<Singleton>>
CellFactoriesFactory
- myCFF: CellFactoriesFactory
<<constructor>> - CellFactoriesFactory ()
+ getCellFactoriesFactory(): CellFactoriesFactory
+ generate(pType: String, pX: int, pY: int): InterfaceCell
<<use>>
<<use>>
<<use>>
Implements
Implements
<<Interface>>
InterfaceCell
+ is(pType: String): boolean
+ canMove(): boolean
+ getId(): int
+ kills(): boolean
Implements
Implements
Implements
<<Abstract>>
Explosion
<<constructor>> # Explosion (pX: int, pY: int)
<<Abstract>> # tiempo(pX: int, pY: int): void
<<@Override>> + is(pType: String): boolean
<<@Override>> + canMove(): boolean
<<@Override>> + getId(): int
<<@Override>> + kills(): boolean
<<Abstract>>
Bomb
<<constructor>> # Bomb (pX: int, pY: int)
<<Abstract>> # tiempo(pX: int, pY: int)
<<@Override>> + is(pType: String): boolean
<<@Override>> + canMove(): boolean
<<@Override>> + getId(): int
<<@Override>> + kills(): boolean
<<Abstract>>
Block
<<constructor>> # Block(pX: int, pY: int)
<<@Override>> + kills(): boolean

# 3.2.3. FACTORY

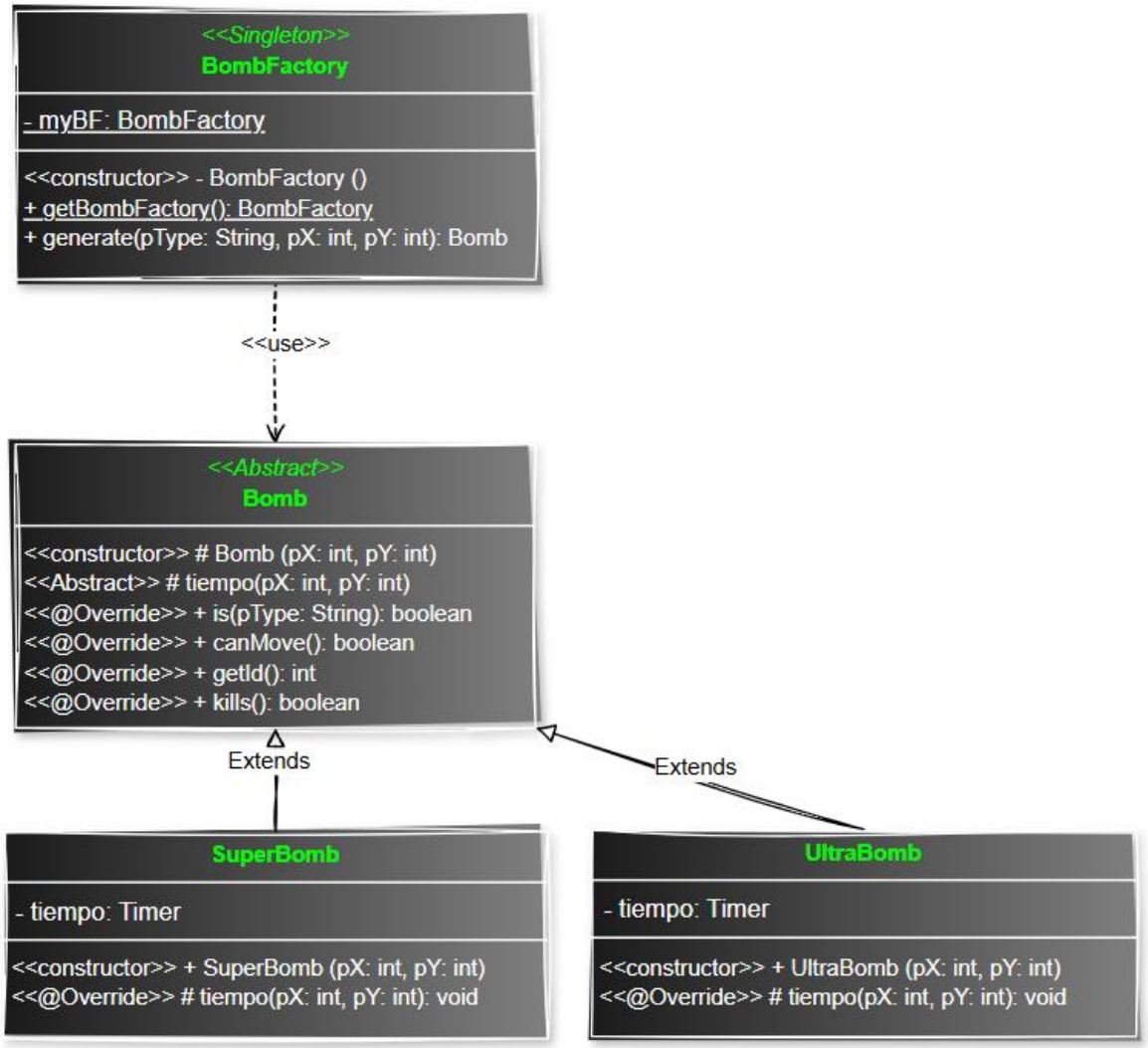
### *3.2.3.1. BLOCK*



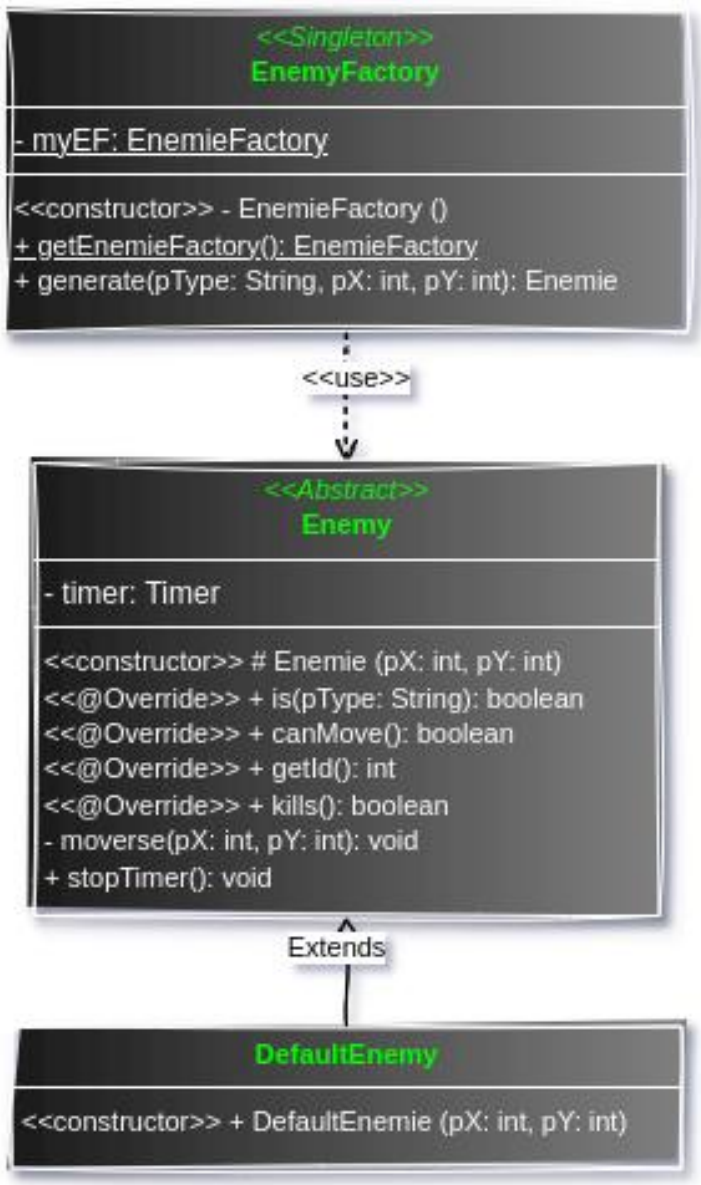
### *3.2.3.2. BOMBERMAN*



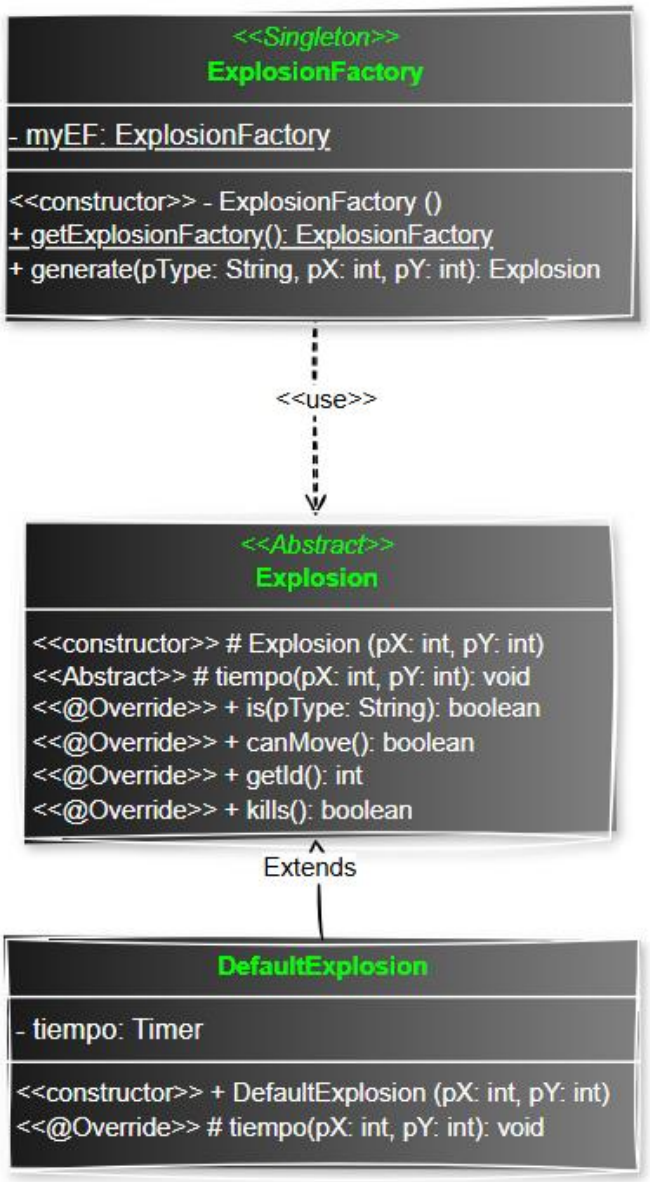
### *3.2.3.3. BOMB*


<<Singleton>>
BombFactory
- myBF: BombFactory
<<constructor>> - BombFactory ()
+ getBombFactory(): BombFactory
+ generate(pType: String, pX: int, pY: int): Bomb
<<use>>
<<Abstract>>
Bomb
<<constructor>> # Bomb (pX: int, pY: int)
<<Abstract>> # tiempo(pX: int, pY: int)
<<@Override>> + is(pType: String): boolean
<<@Override>> + canMove(): boolean
<<@Override>> + getId(): int
<<@Override>> + kills(): boolean
Extends
Extends
SuperBomb
- tiempo: Timer
<<constructor>> + SuperBomb (pX: int, pY: int)
<<@Override>> # tiempo(pX: int, pY: int): void
UltraBomb
- tiempo: Timer
<<constructor>> + UltraBomb (pX: int, pY: int)
<<@Override>> # tiempo(pX: int, pY: int): void

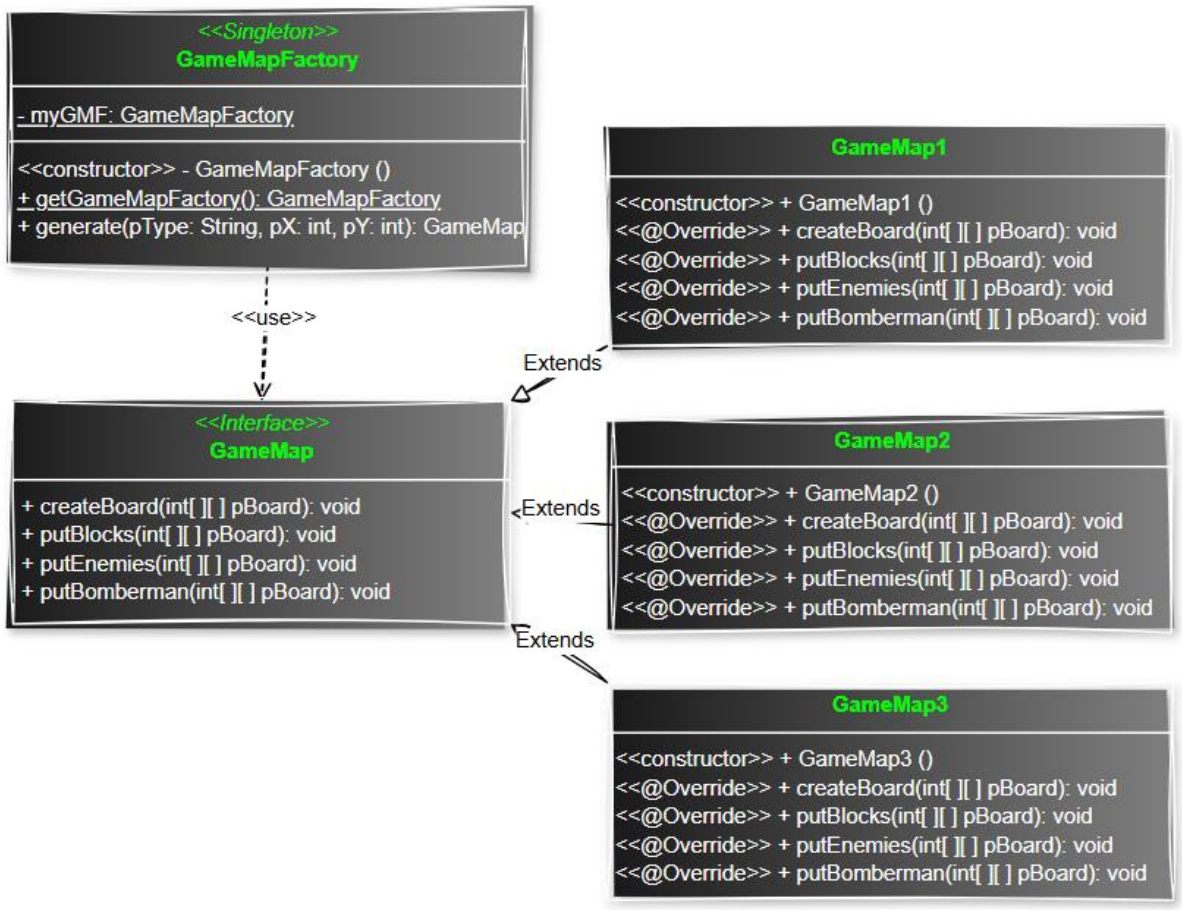
### *3.2.3.4. ENEMY*


<<Singleton>>
EnemyFactory
- myEF: EnemieFactory
<<constructor>> - EnemieFactory ()
+ getEnemieFactory(): EnemieFactory
+ generate(pType: String, pX: int, pY: int): Enemie
<<use>>
<<Abstract>>
Enemy
- timer: Timer
<<constructor>> # Enemie (pX: int, pY: int)
<<@Override>> + is(pType: String): boolean
<<@Override>> + canMove(): boolean
<<@Override>> + getId(): int
<<@Override>> + kills(): boolean
- moverse(pX: int, pY: int): void
+ stopTimer(): void
Extends
DefaultEnemy
<<constructor>> + DefaultEnemie (pX: int, pY: int)

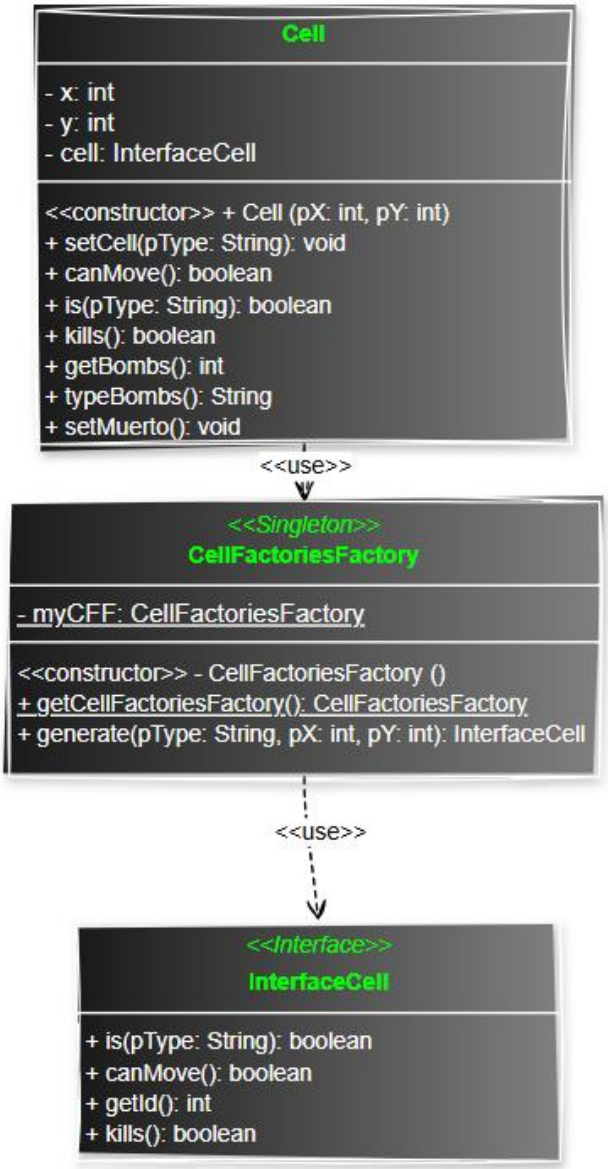
### *3.2.3.5. EXPLOSION*



### *3.2.3.6. GAME MAP*



### *3.2.3.7. CELL FACTORY (FACTORY DE FACTORIES)*



## 3.3. EXPLICACIÓN DETALLADA

### 3.3.1. OBSERVER

Este patrón ha formado parte de nuestro proyecto desde el primer momento, y así continua en este sprint. En nuestro caso, la clase Cell es el observable, y la clase CellView actúa como el observador.

Al crear el tablero, se crea una matriz de celdas y cada una se caracteriza por sus coordenadas únicas. Después, por cada celda, se crea una instancia de CellView, y se añade a sí misma como observer de la casilla correspondiente a través del método GameModel.getGameModel().getCell(pX, pY).addObserver(this).

Cuando ocurre un cambio en el juego, como colocar una bomba o mover al jugador, el modelo (GameModel) modifica el estado de la celda llamando al método setCell(). Dentro de este método, la celda llama a notifyObservers() para informar a todos sus observadores sobre el cambio. El método update() en CellView se ejecuta automáticamente cuando recibe la notificación, y la vista actualiza su la imagen correspondiente llamando al método setImage() con el nuevo estado de la celda.

Aunque GameModel no es ni Observer ni Observable tiene una función muy importante en este patrón ya que coordina los cambios en el tablero, como mover al jugador, colocar bombas o hacer explotar celdas. Cuando el jugador realiza una acción, GameModel actualiza el estado de las celdas afectadas, lo que desencadena la notificación a los observadores (CellView). De forma parecida, TimerModelTool gestiona eventos temporizados, como la explosión de una bomba o el movimiento de enemigos. Cuando uno de estos eventos ocurre, TimerModelTool notifica al modelo, que a su vez actualiza las celdas correspondientes, y estas notifican a sus observadores.

### 3.3.2. STRATEGY

Durante el sprint, el patrón de comportamiento Strategy ha sido útil para la implementación de mapas o para el comportamiento de las celdas de este.

Para la primera de las funciones, cuando el usuario elige el mapa en el que quiere jugar, la clase GameModel guarda en una variable el tipo de mapa que se jugará para en la creación del juego ejecutar el código que tiene el mapa seleccionado. Esta estrategia es la interfaz GameMap implementada por las clases GameMap1, GameMap2 y GameMap3, es decir, los distintos tipos de mapa, los cuales tienen implementados sus

métodos de generación de mapa a su manera para que los diferentes diseños se generen correctamente.

Por otro lado, en cuanto a las celdas, nuestro proyecto está planteado de la forma que cada celda tiene dentro un tipo de celda (Bomberman, bloque, enemigo…), por lo que la clase Cell tiene un atributo que guarda este tipo de celda de tal forma que accede a ella para que sea la celda la que ejecute sus métodos de la forma que esta requiera, ya que, al ser distintos objetos, cada uno tiene su manera de ejecutar los métodos y de comportarse.

### 3.3.3. FACTORY

Durante este sprint se ha implementado el patrón de diseño Factory, cuyo propósito principal es definir una interfaz o clase abstracta para la creación de objetos, permitiendo a las demás subclases decidir qué clase concreta se debe instanciar.

En nuestro código, podemos encontrar el patrón Factory en varias partes del modelo, como en la creación de los bloques (BlockFactory), bombas (BombFactory), enemigos (EnemyFactory), mapas (GameMapFactory) y del bomberman (BombermanFactory). Por ejemplo, la clase BlockFactory se encarga de instanciar los diferentes tipos de bloques según un identificador o parámetro específico. De esta manera, el código cliente no necesita saber qué subclase de Block debe instanciarse, simplemente deja esa responsabilidad al Factory que la heredará.

Este Factory se utiliza dentro de clases como Cell o GameModel para poblar el tablero con los bloques adecuados, sin que estas clases tengan que preocuparse por las instancias concretas.

Esto permite añadir nuevos tipos de bloques sin modificar la lógica del tablero, separar la lógica de creación del resto del modelo y facilitar la reutilización del código en otras partes del juego.

Este patrón trae beneficios como permitir la creación de diferentes tipos de objetos sin modificar el código cliente, facilitar la escalabilidad y el mantenimiento del código…

# 4.GITHUB

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

https://github.com/ikerfernandezmolano/CODIGO_CERO