Implementación del diseño del proyecto BreakOutTec

Paradigmas de Programación

Samuel Marín Soto

2023073212

17 de octubre del 2024

Este código representa fielmente la arquitectura y diseño del sistema especificado, ya que sigue una estructura clara basada en los patrones de diseño previamente definidos y la interacción descrita entre los componentes del sistema.

El primer aspecto central es el uso del **patrón Factory** para manejar las conexiones con los clientes. La clase abstracta *ConexionCliente* define una interfaz común que puede ser extendida tanto por *ConexionClienteJugador* como *ConexionClienteEspectador*. Esto refleja exactamente la especificación en la que los jugadores y espectadores tienen roles distintos, siendo los jugadores capaces de enviar y recibir mensajes, mientras que los espectadores solo reciben actualizaciones del estado del juego. La clase *ConexionClienteFactory* asegura que se puede instanciar adecuadamente el tipo correcto de conexión, basándose en el tipo de cliente (jugador o espectador). Este diseño no solo encapsula la creación de los objetos de manera flexible, sino que también sigue fielmente la lógica establecida para la interacción entre el servidor y los distintos tipos de clientes.

El siguiente componente importante es el uso del **patrón Observer**. La clase *UpdateObserver* implementa la lógica de observación de los clientes, permitiendo que el servidor notifique a todos los clientes conectados sobre las actualizaciones del estado del juego. Esto asegura la sincronización entre el cliente jugador y el cliente espectador, de modo que ambos reciban los mismos cambios en tiempo real. Esta parte del código refleja la necesidad de mantener un juego coordinado, donde los espectadores puedan visualizar el progreso del juego tal como lo ve el jugador, cumpliendo con precisión el requerimiento de comunicación continua entre servidor y clientes.

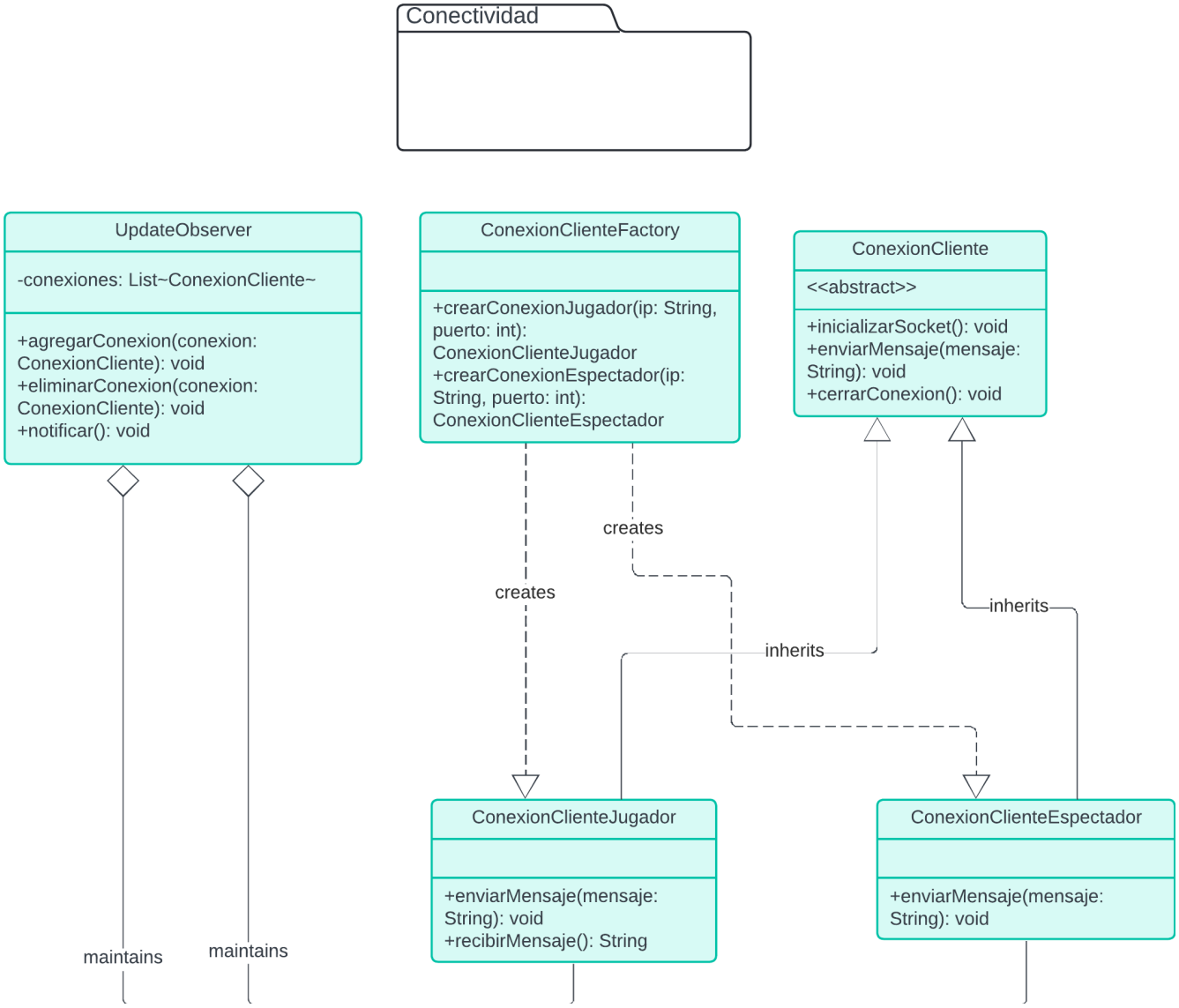
En cuanto a la manipulación de bloques y sus poderes o antipoderes, el código implementa el **patrón Abstract Factory** a través de las clases *AbstractBloqueFactory*, *BloqueBuenoFactory*, y *BloqueMaloFactory*. La *AbstractBloqueFactory* sirve como interfaz para producir efectos abstractos como *TamañoRaqueta*, *VelocidadBola*, *VelocidadRaqueta*, *Bolas*, y *Vidas*. Las fábricas concretas *BloqueBuenoFactory* y *BloqueMaloFactory* implementan la lógica de creación de bloques con poderes o antipoderes, siguiendo exactamente lo que se especificó: *BloqueBuenoFactory* crea efectos positivos como *IncrementarTamañoRaqueta* o *DarVidas*, mientras que *BloqueMaloFactory* genera efectos negativos como *ReducirTamañoRaqueta* o *QuitarVidas*. Este enfoque no solo modulariza la creación de estos objetos, sino que también refleja perfectamente la estructura esperada del sistema, permitiendo que cada fábrica produzca tipos específicos de bloques que pueden afectar el juego de manera diferenciada.

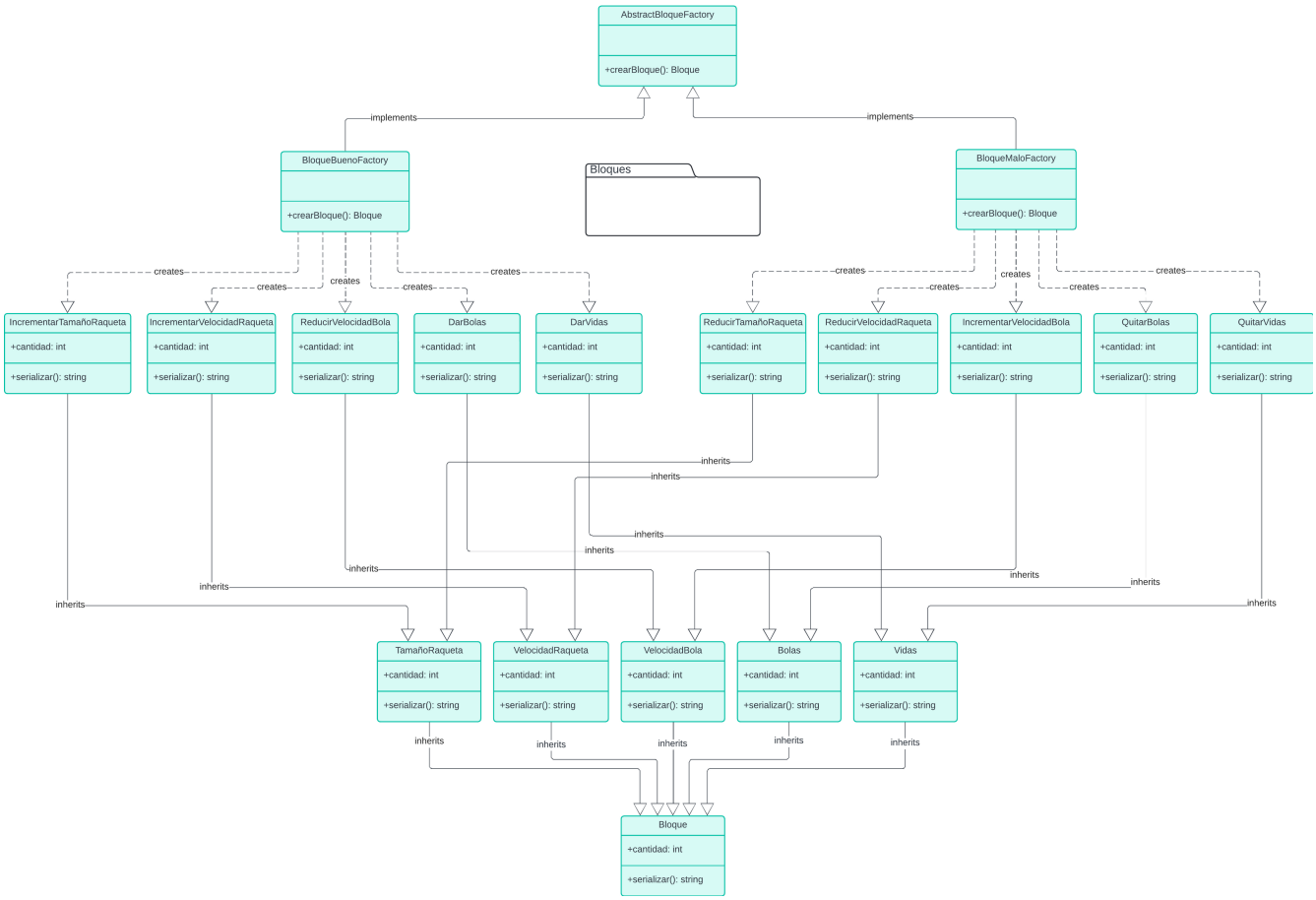
Finalmente, los **productos concretos** de las fábricas, como *IncrementarTamañoRaqueta*, *ReducirTamañoRaqueta*, *DarVidas*, y *QuitarVidas*, están implementados de manera que cumplan con su

propósito al definir una forma estándar de serialización en JSON. Esto asegura que los efectos puedan ser fácilmente comunicados a los clientes como parte de las actualizaciones del estado del juego, algo que es crítico para cumplir con el requisito de enviar información a través de sockets en formato JSON. La capacidad de serializar cada uno de estos efectos es clave para la interacción fluida entre el servidor y los clientes, manteniendo la lógica del juego en sincronía y facilitando la transmisión de los cambios del estado.

En resumen, este código es una representación fiel de la especificación inicial debido a su implementación clara y consistente de los patrones de diseño definidos, su enfoque modular en la creación y manejo de conexiones con los clientes, la sincronización del estado del juego entre múltiples tipos de clientes, y la producción de bloques con efectos específicos que son enviados de manera efectiva en formato JSON para mantener la coherencia del estado del juego. Todo esto asegura que el servidor pueda cumplir su rol de mantener el juego funcionando en armonía entre los clientes jugador y espectador.

Diagramas de Clase y Paquetes:




AbstractBloqueFactory
+crearBloque(): Bloque
implements
implements
BloqueBuenoFactory
+crearBloque(): Bloque
Bloques
BloqueMaloFactory
+crearBloque(): Bloque
creates
creates
creates
creates
creates
creates
creates
creates
creates
creates
creates
IncrementarTamañoRaqueta
+cantidad: int
+serializar(): string
IncrementarVelocidadRaqueta
+cantidad: int
+serializar(): string
ReducirVelocidadBola
+cantidad: int
+serializar(): string
DarBolas
+cantidad: int
+serializar(): string
DarVidas
+cantidad: int
+serializar(): string
ReducirTamañoRaqueta
+cantidad: int
+serializar(): string
ReducirVelocidadRaqueta
+cantidad: int
+serializar(): string
IncrementarVelocidadBola
+cantidad: int
+serializar(): string
QuitarBolas
+cantidad: int
+serializar(): string
QuitarVidas
+cantidad: int
+serializar(): string
inherits
inherits
inherits
inherits
inherits
inherits
inherits
inherits
inherits
TamañoRaqueta
+cantidad: int
+serializar(): string
VelocidadRaqueta
+cantidad: int
+serializar(): string
VelocidadBola
+cantidad: int
+serializar(): string
Bolas
+cantidad: int
+serializar(): string
Vidas
+cantidad: int
+serializar(): string
inherits
inherits
inherits
inherits
inherits
Bloque
+cantidad: int
+serializar(): string