Entrega Final Juego: Block Breaker

Integrantes:

Matias Altamirano
Bastián Jorquera

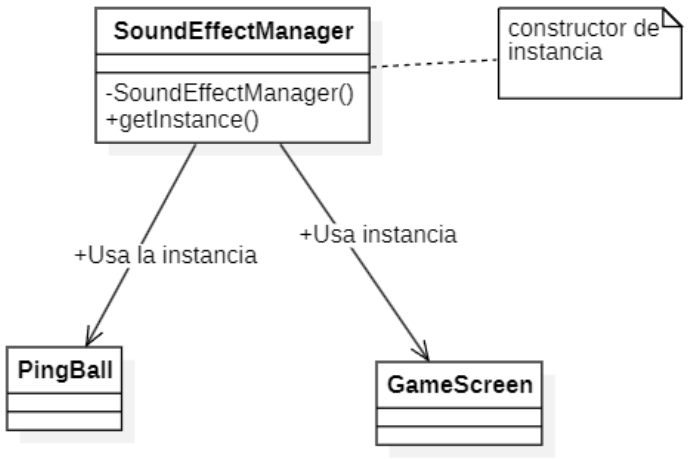
Patrón Singleton

Como bien se sabe este patrón de diseño nos permite crear un constructor global para un atributo específico para usar una instancia de variable a lo largo de la vida de ejecución del programa, y llamarla donde sea necesario. Como candidato para este patrón se consideró la clase "*SoundEffectManager*", la cual genera instancias de sonido cuando la llamaban desde otras clases, así que con este patrón se usará la misma instancia en los distintos efectos de sonido cada vez que se necesite.

Objetivos: Solo usar una instancia de "*SoundEffectManager*", haciendo la lógica más simple y ahorrando código, memoria y recursos.

Se generó una instancia estática con un constructor privado y un método para acceder a la instancia, de este modo el acceso a el atributo quedó protegido, también se modificó la clase "*BlockBreakerGame*" y "*GameScreen*" para que no creen una instancia de sonido, sino que llamen la ya existente.

Diseño UML de clase involucradas con patron Singleton



BlockBreakerGame y GameScreen usan la instancia ya creada.

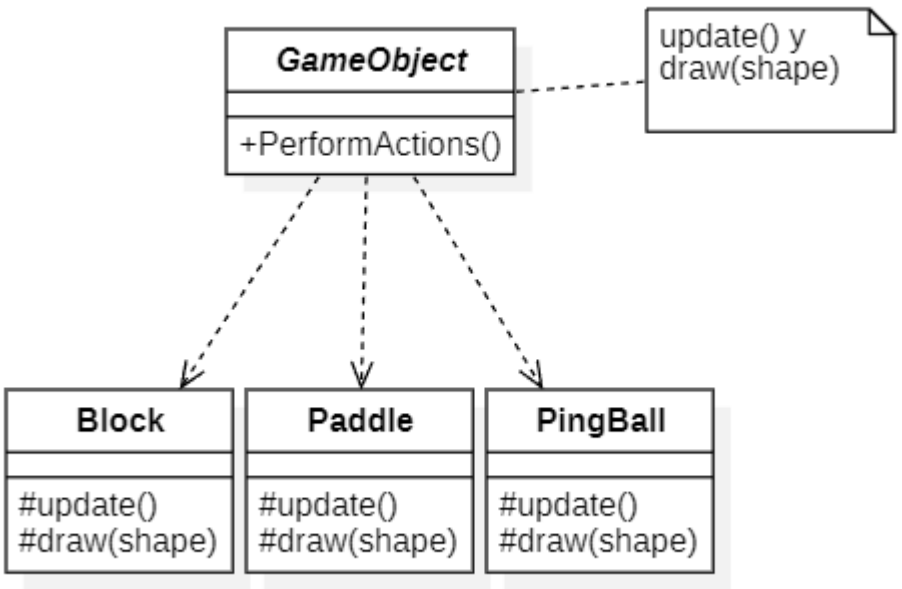
Patrón Template Method

Este patrón de comportamiento define un método plantilla para centralizar un comportamiento en común de las clases, en el caso del juego nos dimos cuenta que los objetos del juego tienen en común los métodos update para definir su posición actual y draw para generarlos, por lo que podemos hacer la plantilla de comportamiento para todos los objetos, además resulta que ya eran hijas de una clase padre, por lo que creemos que es un buen punto para aplicar Template Method.

El patrón permitirá centralizar métodos que se repiten para no usar código y recursos de más, no cambia la lógica ni el comportamiento de los objetos, pero si simplifica el funcionamiento del código.

Se creó la lista de pasos en la clase “GameObject”, se reordenaron los métodos de “draw” y “update” en las clases hijas de esta; “Block”, “Paddle” y “PingBall”, cabe destacar que no tiene implementación actualmente update en la clase “Block”, pero esto no significa que no se cumpla Template Method.

Diseño UML de clase involucradas con patrón Template Method



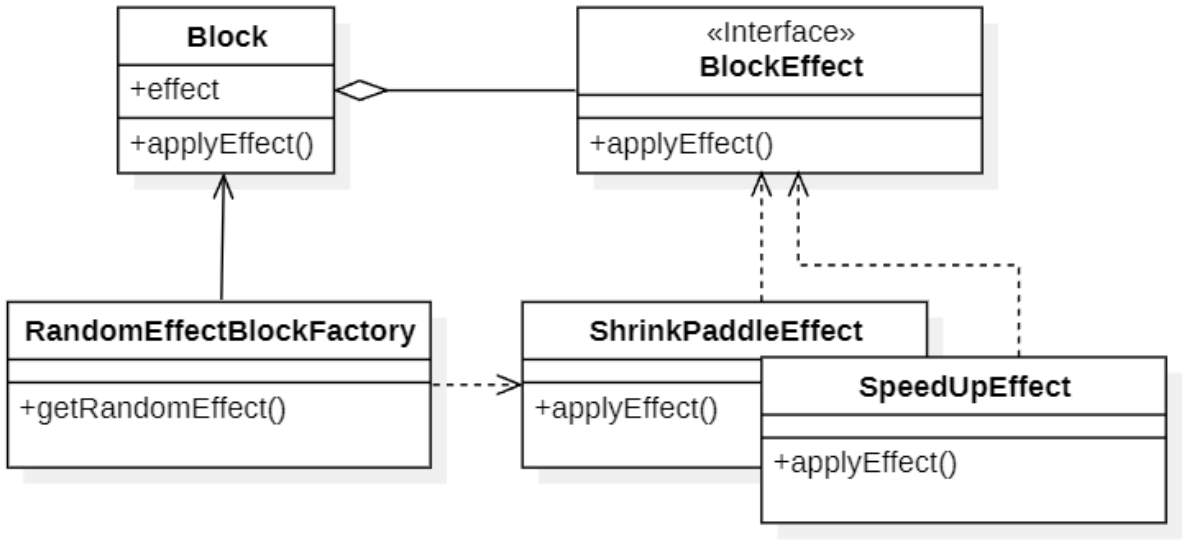
En este UML la clase abstracta GameObject es padre de las otras 3, y tiene un metodo publico con la lista de acciones que deben hacer las clases hijas.

Patrón Strategy

La problemática se origina al intentar implementar distintos efectos al romper los bloques (aumentar tamaño de Paddle, variar la velocidad de Ball, entre otros). Se necesita implementar variadas funciones que en su estructura serían muy semejantes entre sí. Además, pensando en el crecimiento del software a futuro, se necesita que la implementación de estas funciones sea fácil de modificar y mantener, asimismo, al aplicar el patrón Strategy se ahorra en múltiples verificaciones que podrían dificultar las legibilidad del código.

En este caso, se creó la interfaz "*BlockEffect*" con el método *applyEffect(PingBall ball, Paddle paddle)* el cual es implementado por clases que sobreescribe el método como por ejemplo, "*shrinkPaddleEffect*", el cual modifica el tamaño del Paddle. Así, simplemente se crearía una clase que implemente "*BlockEffect*", el método se configura con el efecto que se desea lograr y listo, un nuevo efecto se ha creado. Estos métodos son utilizados por los constructores del objeto "*Block*" que otorgan un valor de *efecto* a un bloque generado al iniciar el juego. El efecto será aplicado una vez la pelota colisione con el bloque siendo gestionado por la Interfaz "*Collidable*", específicamente con el método *onCollision()*.

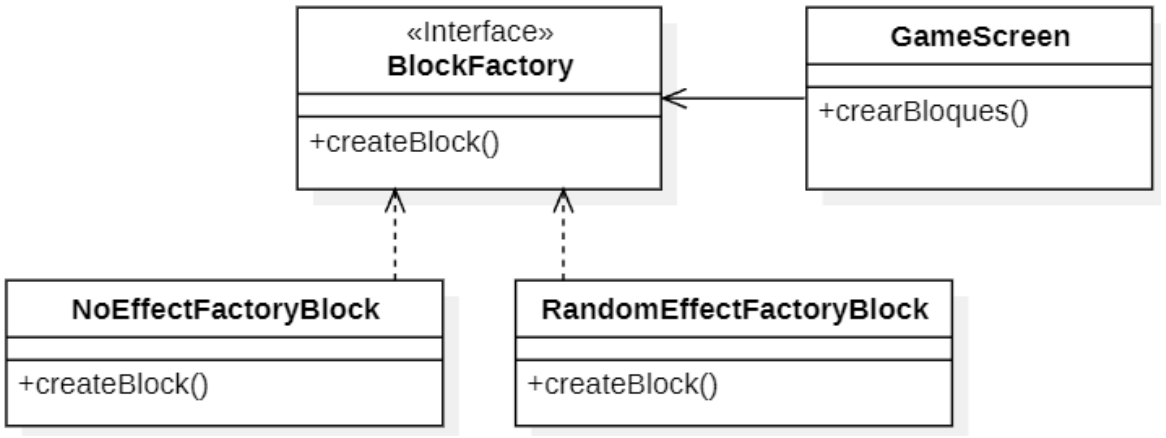
Diseño UML de clase involucradas con patrón Strategy



Patrón de Creación: Abstract Factory

Como se mencionó anteriormente, el patrón Strategy se utilizó para implementar distintas variaciones de un algoritmo que será utilizado por una clase en específico. Pero con esto nace un nuevo problema, ¿cuál será la clase contexto que va a invocar los bloques con distinto efecto? Para ello se creó la interfaz "*BlockFactory*", el cual contiene un constructor que permite generar bloques con un nuevo atributo llamado "efecto". Esta interfaz es implementada por dos nuevas clases: "*NoEffectBlockFactory*" que genera bloques con el atributo "*efecto*" como null; y la clase "*RandomEffectBlockFactory*", la cual contiene un método que otorga un efecto aleatorio (proveniente de los "*BlockEffect*" mencionados anteriormente) al nuevo bloque a generar. Los bloques son generados por la clase que gestiona el juego: "*GameScreen*", la cual utiliza el método *crearBloques()* para llamar al método *createBlock()* y generar bloques de diversos tipos.

Diseño UML de clase involucradas con Patrón Abstract Factory



Función extra

Un juego sin menú inicial ni menú de pausa, es como un libro sin portada ni signos de puntuación (punto seguido, punto aparte). No hay forma de saber cómo se llama o imaginar de qué podría tratar el libro y tampoco podrías encontrar un punto adecuado en la narración para poder tomar un descanso de la lectura. Es importante dar al usuario una experiencia previa al juego, guiarlo en su inmersión a este y darle momentos en los que puede descansar. Para ello la implementación de ambos Menús es fundamental, tanto como para la experiencia del usuario, como para el desarrollo del juego a nivel de código.

En el proceso de efectuar estos cambios, se encontró otra problemática, el manejo de las pantallas (screens). En el juego base "*BlockBreaker*", la clase *"main" ("BlockBreakerGame")* es la que gestiona la pantalla y las instancias del juego en general, por lo que, si se quiere implementar un menú de inicio, se debe separar la gestión de pantallas de la clase principal. Para ello se creó la clase "*MainMenuScreen*", la cual se encarga de renderizar el *Menú Inicial*; y otra clase llamada "*GameScreen*", la cual se encarga de renderizar la pantalla en la cual se juega. De esta forma la clase main "*BlockBreakerGame*", solo se encarga de Iniciar la clase "*MainMenuScreen*" y queda libre de cualquier otra responsabilidad.

Para resumir lo anterior, se creó la clase "*MainMenuScreen*" la cual inicia un *Menú Inicial*. En este último se despliega el nombre del juego, indica que presionando cualquier tecla el juego dará inicio y en la zona inferior entrega las instrucciones en cuanto al uso de teclas. Se creó la clase "*GameScreen*" que se encarga de iniciar el juego y en la cual se accede al *Menú de Pausa* presionando la tecla "*Esc*" (escape). El menú despliega 3 opciones: Continuar (el juego), Volver al Menú (vuelve al menú inicial) y Cerrar el Juego (cierra el juego). Junto a lo anterior, a nivel de código se logra aliviar la carga del "*main*", la cual hasta antes de los cambios cargaba con mucha responsabilidad.

En conclusión, con los cambios realizados, se logra ofrecer una experiencia de usuario más amigable y completa. A nivel de código, el software es cada vez más estructurado, escalable y modular, lo que permite un mejor mantenimiento, una mejor capacidad de desarrollo y una mayor vida útil.

Link de GitHub

https://github.com/BastianJorquera/BlockBreaker/tree/Entrega-Final