# Tarea Patrones de Diseño

**Integrantes:**

Savier Acosta Suquitana

Olivier León Marquez

Rommel Zamora Wong

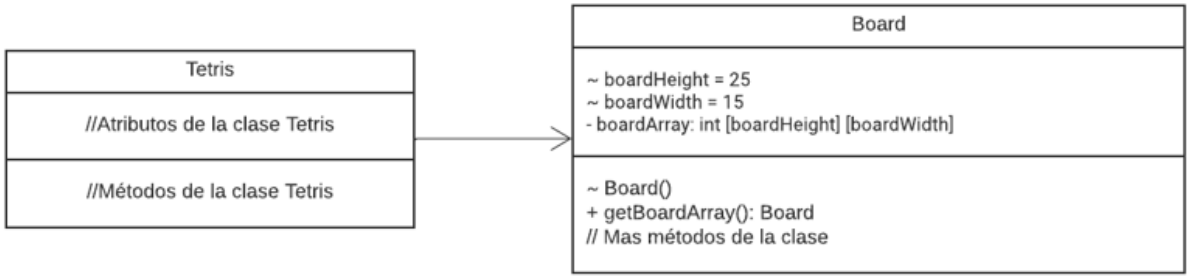
- **Patrón Singleton:**

**Objetivo:** Implementar el patrón Singleton en la clase ‘’Board’’ para tener una única instancia del tablero en el contexto del juego ‘’Tetris’’.

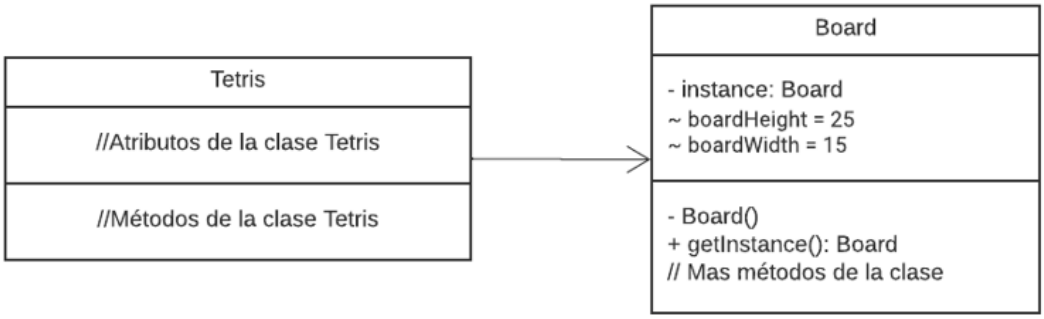
**Motivación:** Para este repositorio nos percatamos de que es necesaria la implementación de este patrón de diseño creacional para la clase ‘’Board’’, esto debido a que sabemos que en el contexto del juego ‘’Tetris’’ no es necesario el tener más de una instancia de ‘’Board’’ ya que de por si el usuario solo interactúa con un único tablero y no con varios. Además, sabemos que el tablero es una clase que interactúa de manera dinámica con muchas otras clases, entonces al tener una única instancia evitamos pasar la instancia del tablero constantemente entre otras clases.

La solución que se ha propuesto para la aplicación del patrón Singleton en la clase ‘’Board’’ es la siguiente:

**Diseño Inicial:**




**Rediseño posterior:**



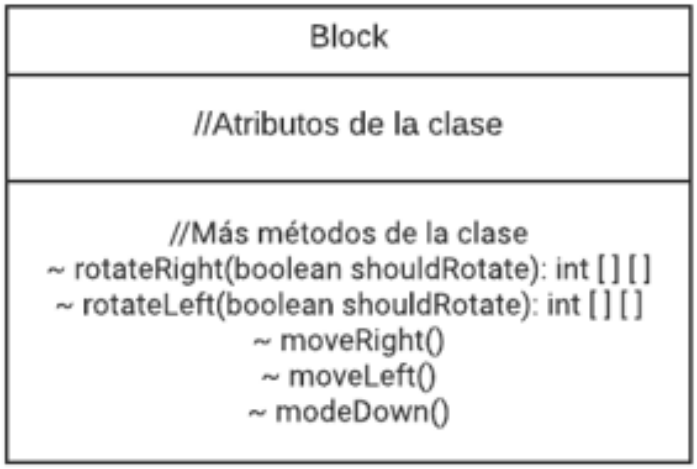
- **Patrón Strategy:**

**Objetivo:** Implementar el patrón Strategy en la clase ‘’Block’’ para tener distintos algoritmos que permitan realizar acciones distintas para el bloque.

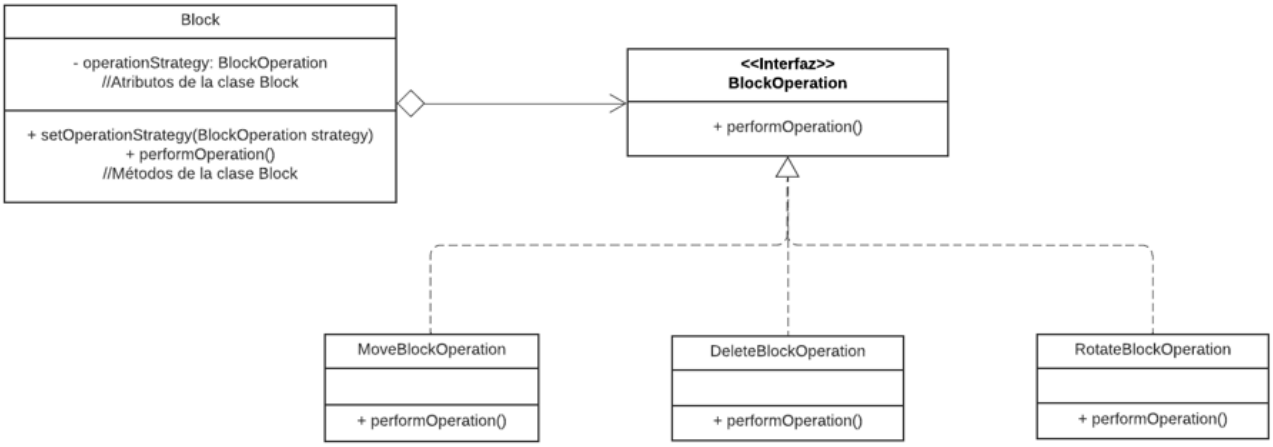
**Motivación:** Para este repositorio creemos que es necesario el implementar el patrón Strategy para la clase ‘’Block’’ ya que sabemos que esta misma entidad tiene diferentes comportamientos (algoritmos) dentro de la misma clase. Lo ideal sería separar estos comportamientos aplicando Strategy y de esta forma la flexibilidad y mantenibilidad del código se ve mejorada debido a que diferentes estrategias pueden ser agregadas, modificadas o reemplazadas sin modificar la clase ‘’Block’’.

La solución que se ha propuesto para la aplicación del patrón Strategy en la clase ‘’Block’’ es la siguiente:

**Diseño inicial:**



**Rediseño posterior:**



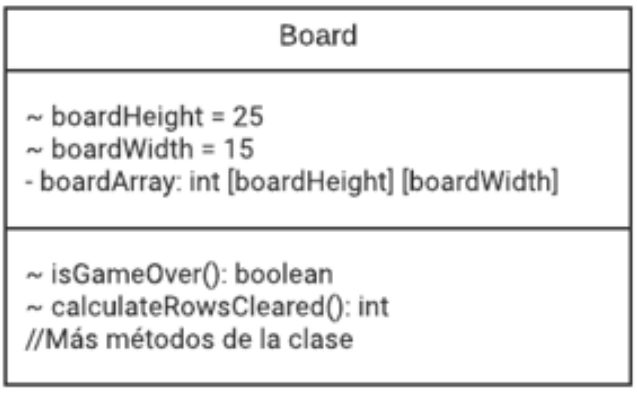

- **Patrón Observer:**

**Objetivo:** Implementar el patrón Observer dentro del código para lograr un desacoplamiento entre las clases involucradas.

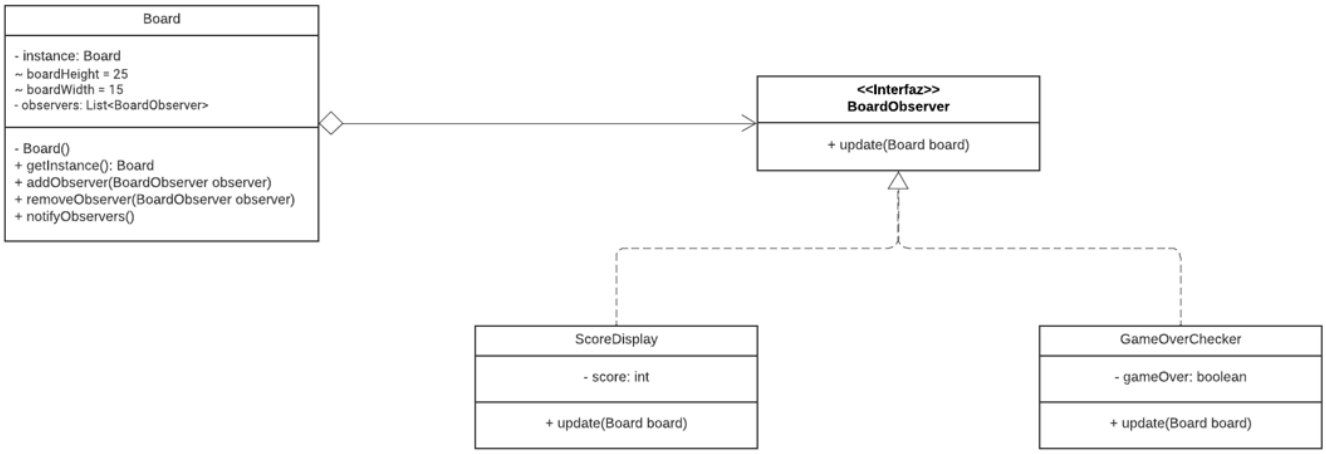
**Motivación:** Para este repositorio sería bastante beneficioso implementar el patrón Observer para la clase ‘’Board’’. Además, es posible separar ciertas responsabilidades de ciertas clases en otras para distribuir de mejor manera las funcionalidades de cada clase, por ejemplo, aplicando el patrón Observer podemos crear ciertas clases que actúen como observadoras de la clase ‘’Board’’ como ‘’ScoreDisplay’’ o ‘’GameOverChecker’’.

La solución que se ha propuesto para la aplicación del patrón Observer en la clase ‘’Board’’ es la siguiente:

**Diseño inicial:**



**Diseño posterior:**



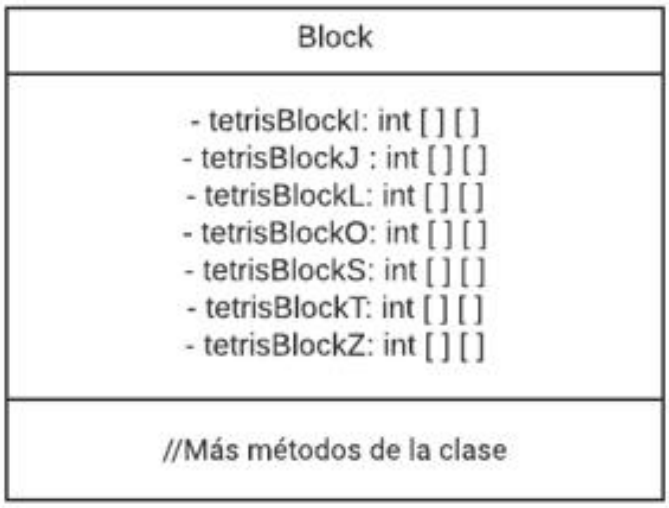

- **Patrón Factory Method:**

**Objetivo:** Implementar el patrón Factory Method para representar las distintas formas que puede tomar la clase ‘’Block’’ dentro del programa.

**Motivación:** Sabemos que en la clase ‘’Block’’, un bloque puede tener distintas representaciones dependiendo de cual requiera el programa, pero si mantenemos el código de esta clase como está ahora y luego quisiéramos agregar otros tipos de bloques es muy probable tener que modificar nuevamente ciertas partes de la clase ‘’Block’’ para introducir la nueva lógica del nuevo bloque. Sin embargo, al implementar el patrón Factory Method podemos facilitar la extensibilidad del código al momento de crear otras entidades y centraríamos la lógica de creación dentro del patrón.

La solución que se ha propuesto para la aplicación del patrón Factory Method en la clase ‘’Block’’ es la siguiente:

**Diseño inicial:**



**Rediseño posterior:**