Es un juego de rompecabezas clásico, donde las piezas con formas geométricas caen desde la parte superior de la pantalla. El objetivo del jugador es mover y rotar estas piezas para encajarlas de manera que llenen filas horizontales completas. Cuando se completa una fila, desaparece, y los bloques por encima caen, otorgando puntos al jugador. El juego se vuelve más rápido con el tiempo, y el jugador pierde cuando las piezas se acumulan hasta la parte superior de la pantalla.

1. Single Responsibility Principle (SRP) - Principio de Responsabilidad Única

Significa: Cada clase debe tener una única responsabilidad o razón para cambiar.

Cómo lo cumple el código:

La clase Board maneja todo lo relacionado con el tablero del juego.

La clase Shape solo se encarga de las formas del Tetris (cómo se mueven, rotan, etc.).

WindowGame maneja la ventana del juego y el ciclo principal.

Ejemplo: Si un día decides cambiar el diseño del tablero, solo necesitarías modificar la clase Board, ya que está encargada de eso.

2. Open/Closed Principle (OCP) - Principio Abierto/Cerrado

Significa: El código debe estar abierto para extenderse (agregar nuevas funciones) pero cerrado para modificarse (no tener que cambiar el código existente).

Cómo lo cumple el código:

No está completamente claro en este caso. Si quisieras agregar nuevas características (como nuevos tipos de bloques), tendrías que modificar el código de Shape y Board, lo cual rompe un poco este principio.

Ejemplo de mejora: Podrías crear una clase abstracta o interfaz para las formas del Tetris, y luego extender esa clase cada vez que quieras agregar una nueva forma sin modificar la clase base.

3. Liskov Substitution Principle (LSP) - Principio de Sustitución de Liskov

Significa: Si tienes una clase base (ejemplo: una clase general para las formas del Tetris), deberías poder reemplazarla por cualquier subclase (por ejemplo, una nueva forma de Tetris) sin que el programa falle.

Cómo lo cumple el código: No se ve mucho este principio en tu código, ya que no hay una estructura clara de herencia o sustitución entre clases. Pero, si lo hubiera, deberías asegurarte de que cualquier clase nueva que heredes funcione sin problemas.

4. Interface Segregation Principle (ISP) - Principio de Segregación de Interfaces

Significa: Las clases no deben depender de interfaces que no usan.

Cómo lo cumple el código:

En tu código, Board implementa varios interfaces (KeyListener, MouseListener, MouseMotionListener), lo que podría romper este principio porque quizás Board no debería manejar tantas cosas a la vez.

Ejemplo de mejora: Podrías crear clases separadas para manejar los eventos del teclado y ratón, dejando que Board solo se encargue de la lógica del juego.

5. Dependency Inversion Principle (DIP) - Principio de Inversión de Dependencias

Significa: Las clases de alto nivel no deben depender de clases concretas de bajo nivel. Ambas deben depender de abstracciones.

Cómo lo cumple el código:

En tu código, Board depende directamente de Shape. Para mejorar esto, podrías hacer que Board dependa de una interfaz o clase abstracta, y no de una implementación concreta como Shape.

Ejemplo de mejora: Podrías tener una interfaz llamada IShape que defina las operaciones que una forma debe tener (moverse, rotarse, etc.), y Board dependería de IShape, no de Shape. Así, si un día creas un nuevo tipo de forma, no tendrías que cambiar la clase Board.