Guión para Video Explicativo: Sistema de Análisis de Caballos en Conflicto

Introducción

[Pantalla de Título: "Caballos en Conflicto - Programación Orientada a Objetos en Ajedrez"]

Narrador: ¡Hola! Bienvenidos a este tutorial donde combinaremos el fascinante mundo del ajedrez con la Programación Orientada a Objetos. Crearemos un sistema que analiza conflictos entre caballos en un tablero de ajedrez.

[Mostrar tablero de ajedrez con varios caballos posicionados]

Narrador: Los caballos son las piezas más únicas del ajedrez: se mueven en forma de "L" y pueden "saltar" sobre otras piezas. Nuestro sistema determinará cuándo un caballo está "en conflicto", es decir, bajo amenaza de otro caballo.

Reglas del movimiento del caballo:

- **Forma de L:** 2 casillas en una dirección + 1 casilla perpendicular
- **8 movimientos posibles:** desde cualquier posición válida
- **Notación algebraica:** columnas A-H, filas 1-8 (ejemplo: B7, C5)

```
Ejemplo de entrada: B7,C5,E2,H7,G5,F6
Salida esperada: Análisis de conflictos entre cada par de caballos
```

Narrador: Nuestro objetivo es crear un sistema que convierta notación algebraica a coordenadas, calcule movimientos posibles y detecte amenazas automáticamente.

Desarrollo Parte 1: La Estructura Coordinate

[Mostrar código de la estructura Coordinate]

Narrador: Comenzamos con una **estructura Coordinate** que representa una posición en el tablero:

Narrador: Aquí vemos un concepto importante: **struct vs class**:

¿Por qué usar struct?

- **Tipo de valor:** se almacena directamente en memoria, no como referencia
- **Inmutabilidad:** ideal para coordenadas que no cambian
- **Rendimiento:** menos overhead de memoria que las clases
- **Comparación:** podemos sobrescribir Equals() para comparar posiciones

Método Equals() personalizado: Permite comparar dos coordenadas directamente, esencial para detectar si un caballo amenaza la posición de otro.

Narrador: Esta estructura encapsula las coordenadas X,Y y proporciona métodos para compararlas. Es la base matemática de todo nuestro sistema.

Desarrollo Parte 2: La Clase Knight

[Mostrar código de la clase Knight]

Narrador: La clase **Knight** es el corazón de nuestro sistema. Representa un caballo con su posición y capacidades:

Narrador: Esta clase maneja dos representaciones de la misma información:

Doble representación:

- **PositionAlgebraic:** notación humana (B7, C5, etc.)
- **X, Y:** coordenadas numéricas para cálculos (2,7), (3,5), etc.
- **Conversión automática:** el constructor maneja la transformación
- **Encapsulación:** propiedades de solo lectura protegen la integridad

Conversión Algebraica a Coordenadas

[Mostrar método ConvertAlgebraicToCoordinates]

Narrador: Este método realiza una **traducción matemática**:

```
Ejemplos de conversión:
A1 → (1,1) | B7 → (2,7) | H8 → (8,8)
Fórmula: X = letra - 'A' + 1, Y = número - '0'
```

Validación integrada: Verifica que las coordenadas estén dentro del tablero (A-H, 1-8) y lanza excepciones si son inválidas.

Cálculo de Movimientos Posibles

[Mostrar método GetPossibleMoves]

Narrador: El método más importante es **GetPossibleMoves()**, que calcula las 8 posiciones que puede atacar un caballo:

Narrador: Este algoritmo usa **vectores de desplazamiento** para modelar el movimiento en L:

Vectores de movimiento:

- **deltaX, deltaY:** arrays paralelos que definen los 8 movimientos
- **Patrón sistemático:** (+2,+1), (+2,-1), (-2,+1), (-2,-1), (+1,+2), (+1,-2), (-1,+2), (-1,-2)
- **Validación de límites:** solo agrega movimientos dentro del tablero
- **Estructura de datos:** devuelve List para flexibilidad

```
Ejemplo visual desde B7 (2,7):
Puede atacar: A5, C5, D6, D8
(Los otros 4 movimientos salen del tablero)
```

Desarrollo Parte 3: La Clase ChessBoard

[Mostrar código de ChessBoard]

Narrador: La clase **ChessBoard** coordina todo el análisis. Actúa como el "cerebro" que maneja múltiples caballos:

Narrador: Esta clase demuestra **procesamiento de entrada** robusto:

Características del parsing:

- **Split() por comas:** convierte string en array de posiciones
- **Trim():** elimina espacios extras automáticamente
- **Try-catch:** maneja posiciones inválidas sin crash
- **Colección dinámica:** List crece según la entrada

Análisis de Conflictos

[Mostrar método FindThreatenedKnights]

Narrador: El método **FindThreatenedKnights()** implementa la lógica de detección de amenazas:

Narrador: Este algoritmo realiza una **comparación exhaustiva**:

Algoritmo paso a paso:

1. Obtiene todos los movimientos posibles del caballo atacante

2. Para cada otro caballo en el tablero
3. Convierte su posición a Coordinate
4. Compara con cada movimiento posible usando Equals()
5. Si hay coincidencia, lo agrega a la lista de amenazados

Optimizaciones implementadas:

- **Auto-exclusión:** un caballo no puede amenazarse a sí mismo
- **Break temprano:** sale del loop cuando encuentra amenaza
- **Reutilización:** usa el método GetPossibleMoves() existente
- **Separación de responsabilidades:** detecta amenazas, no las muestra

Desarrollo Parte 4: Formato de Salida y Interfaz

[Mostrar método AnalyzeConflicts]

Narrador: El método **AnalyzeConflicts()** produce la salida en el formato específico requerido:

Narrador: Este método maneja el **formato de salida específico**:

Características del formato:

- **Inversión de coordenadas:** muestra "7B" en lugar de "B7" (fila + columna)
- **Múltiples conflictos:** cada uno en línea separada con "Conflicto con"
- **Sin conflictos:** línea vacía después del "=>"
- **Formato consistente:** misma estructura para todos los caballos

Desarrollo Parte 5: El Programa Principal

[Mostrar código del Main]

Narrador: El programa principal demuestra **arquitectura modular**:

Narrador: Este diseño demuestra **principios de arquitectura limpia**:

Separación de responsabilidades:

- **Main():** interfaz de usuario y control de flujo
- **RunTestCase():** pruebas automáticas
- **ProcessKnightPositions():** lógica de procesamiento
- **ChessBoard:** análisis de ajedrez

Flujo de datos: Entrada de texto → Parsing → Objetos Knight → Análisis → Salida formateada

Demostración Práctica

[Mostrar ejecución del programa]

Narrador: Veamos el sistema con el caso de prueba: **B7,C5,E2,H7,G5,F6**

Análisis paso a paso:

- **B7 (2,7):** Puede atacar A5, C5, D6, D8 → Amenaza a C5
- **C5 (3,5):** Puede atacar A4, A6, B3, B7, D3, D7, E4, E6 → Amenaza a B7
- **E2 (5,2):** Puede atacar C1, C3, D4, F4, G1, G3 → No amenaza a nadie
- **H7 (8,7):** Puede atacar F6, F8, G5 → Amenaza a F6
- **G5 (7,5):** Puede atacar E4, E6, F3, F7, H3, H7 → Amenaza a H7
- **F6 (6,6):** Puede atacar D5, D7, E4, E8, G4, G8, H5, H7 → Amenaza a G5 y H7

```
Salida esperada:
Analizando Caballo en 7B => Conflicto con 5C
Analizando Caballo en 5C => Conflicto con 7B
Analizando Caballo en 2E =>
Analizando Caballo en 7H => Conflicto con 6F
Analizando Caballo en 5G => Conflicto con 7H
Analizando Caballo en 6F => Conflicto con 7H
Conflicto con 5G
```

Narrador: Este ejemplo demuestra **detección automática** de amenazas mutuas y múltiples conflictos por caballo.

Conclusión

Narrador: Este sistema demuestra cómo la programación puede modelar reglas complejas del mundo real. La combinación de matemáticas, lógica y POO crea soluciones elegantes y extensibles.

Narrador: El ajedrez, con sus reglas precisas y posiciones complejas, es un excelente dominio para aplicar conceptos de programación estructurada y orientada a objetos.

[Pantalla final con tablero de ajedrez mostrando amenazas detectadas]

Narrador: ¡Gracias por acompañarnos en este recorrido por el ajedrez computacional y la Programación Orientada a Objetos!