# Inicio

Qué gusto que estés aprendiendo DDD conmigo, y excelente elección de dominio con el ajedrez. Para comenzar con buen pie, te recomiendo seguir este **orden de implementación**, que te ayudará a **pensar como DDDer** desde el inicio sin abrumarte:

**✅ Paso 1: Comprender el *core domain* — ¿Qué es realmente una partida de ajedrez?**

Antes de escribir código, hazte estas preguntas DDD-style:

* ¿Qué **objetos o conceptos** existen en una partida?
* ¿Quién es el **actor principal**? ¿Qué hace?
* ¿Qué **reglas** no pueden romperse?
* ¿Qué evento importante puede ocurrir en la partida? (jaque mate, tablas, etc.)

**🎯 Paso 2: Empezar por el Agregado Raíz: PartidaDeAjedrez**

**¿Por qué?**

Porque DDD se centra en el **corazón del comportamiento del dominio**, y ese corazón es el **agregado raíz**.

La PartidaDeAjedrez:

* Es el punto de entrada para registrar movimientos
* Conoce el tablero, los jugadores y el estado del juego
* Es responsable de mantener **la coherencia y reglas** del juego

**🧱 Paso 3: Acompañar con los elementos de apoyo**

Luego de tener una idea clara del agregado raíz, agregamos los componentes que le dan forma:

**🔸 Entidades:**

* Jugador (con identidad)
* Pieza (si decides que cada una tiene identidad propia, como una reina que se promociona, etc.)

**🔸 Objetos de Valor:**

* Movimiento (desde, hacia, tipo de pieza)
* Coordenada o Posicion (como "e4")
* Color (BLANCO, NEGRO)

**🔸 Value Object vs Entidad**

* Las piezas pueden ser **VOs** si solo importa su tipo y posición (más simple)
* Pueden ser **entidades** si quieres control más detallado (historia, movimientos individuales, etc.)

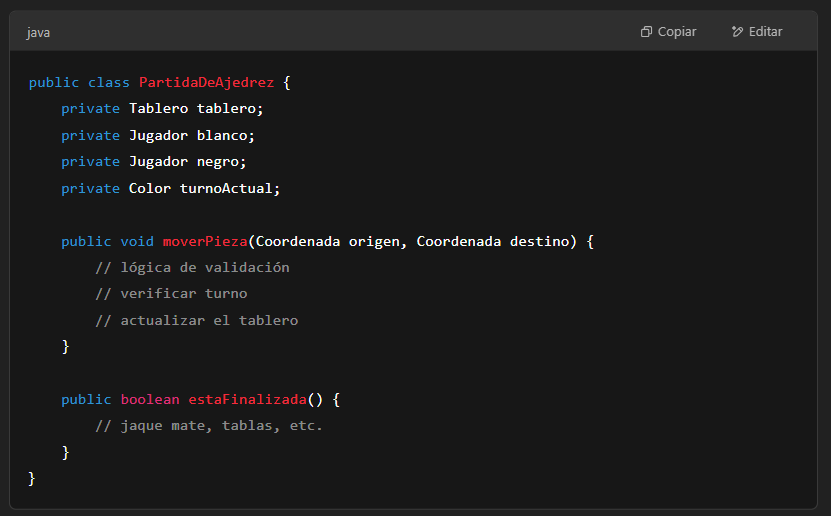
**🟡 Alternativa: empezar por Tablero y Pieza**

Puedes hacerlo, pero aquí está la diferencia:

* Si empiezas por el **tablero**, trabajas con “datos” primero
* Si empiezas por **PartidaDeAjedrez**, trabajas con “comportamiento” desde el principio (más alineado a DDD)

**✅ Mi sugerencia como mentor DDD:**

**Empieza por el Agregado PartidaDeAjedrez**, incluso si lo haces vacío al inicio, y luego “construye hacia adentro”:



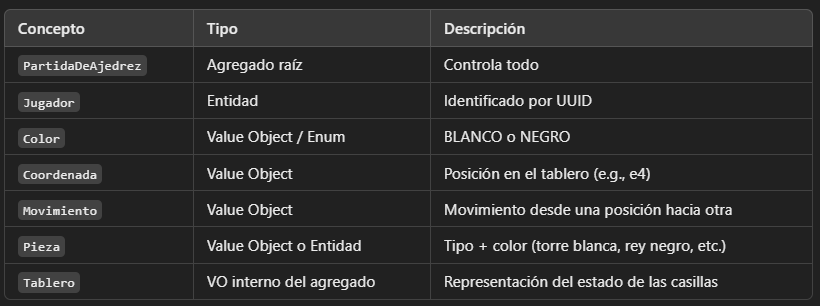
# Definir Dominio

**🧠 Paso 1: Definir conceptos clave del dominio**

Queremos un sistema que pueda:

* Crear una partida entre dos jugadores
* Controlar los turnos
* Mover piezas
* Detectar el fin de la partida (jaque mate o tablas)
* Consultar el estado actual del tablero y del juego

Así que los conceptos mínimos que necesitamos son:



**✅ Paso 2: Esquema base del Agregado PartidaDeAjedrez**

Vamos a crear una primera versión del **agregado raíz**, sencilla pero ya orientada a la lógica de negocio.



**🔍 ¿Qué contiene este agregado?**

✅ Control del turno  
✅ Validación de la pieza y su movimiento  
✅ Control de finalización por jaque mate (simplificado por ahora)  
✅ Orquestación del dominio: el tablero no actúa solo, todo pasa por la partida

**🧱 Siguiente paso: construir las piezas internas**

Desde aquí, lo natural sería crear:

* Color → Enum sencillo
* Jugador → Entidad simple
* Coordenada → Value Object
* Pieza → Clase con tipo y lógica de movimiento
* Tablero → Clase que representa el estado actual

# Construyendo la clase Color, Jugador y Coordenada

## 🎨 1. Color (Value Object / Enum)

Define los colores posibles en ajedrez: BLANCO y NEGRO.

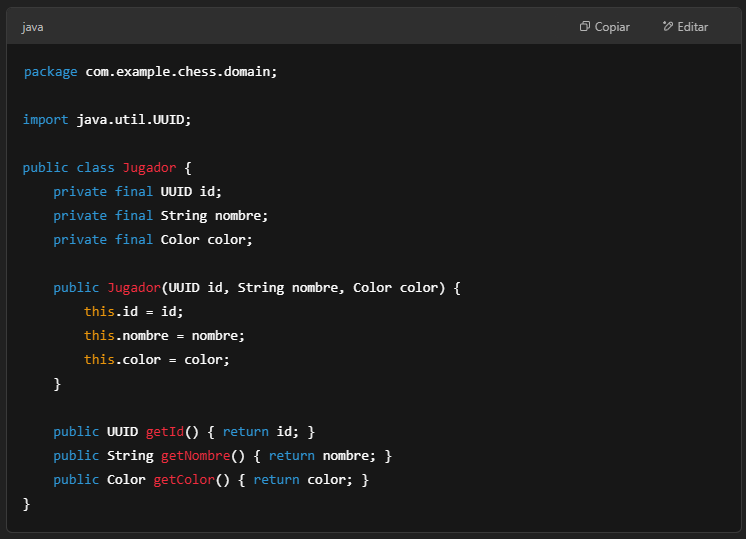


**🔍 ¿Por qué es importante?**

* Se usa en piezas, jugadores y turnos.
* El método oponente() permite cambiar turno fácilmente.

## 🧍‍♂️ 2. Jugador (Entidad)

Una entidad simple que representa a un jugador con identidad.

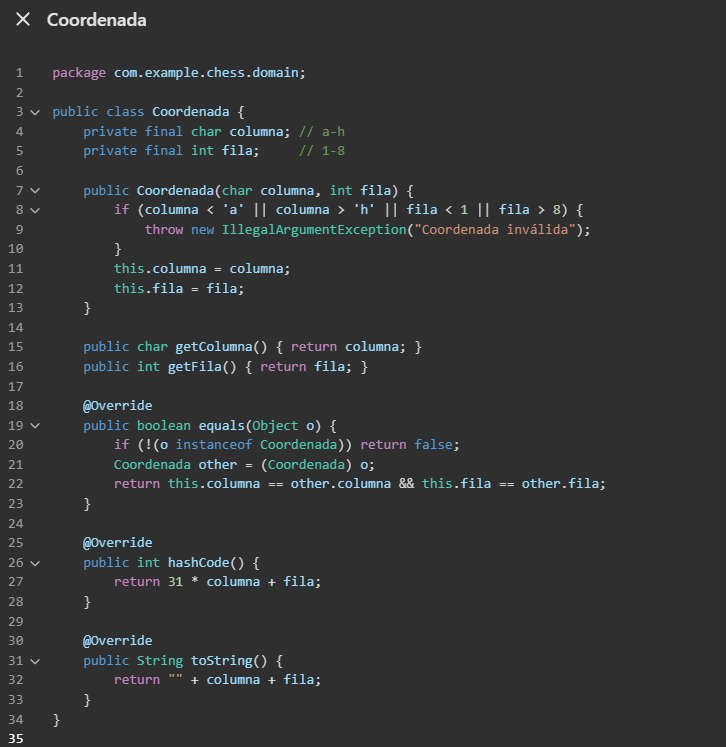


**🔍 ¿Por qué es una entidad?**

* Tiene un **UUID** como identidad única.
* Puede haber dos jugadores con el mismo nombre, pero no con el mismo ID.

## 📍 3. Coordenada (Value Object)

Representa una casilla del tablero, como “e4”, “b7”, etc.

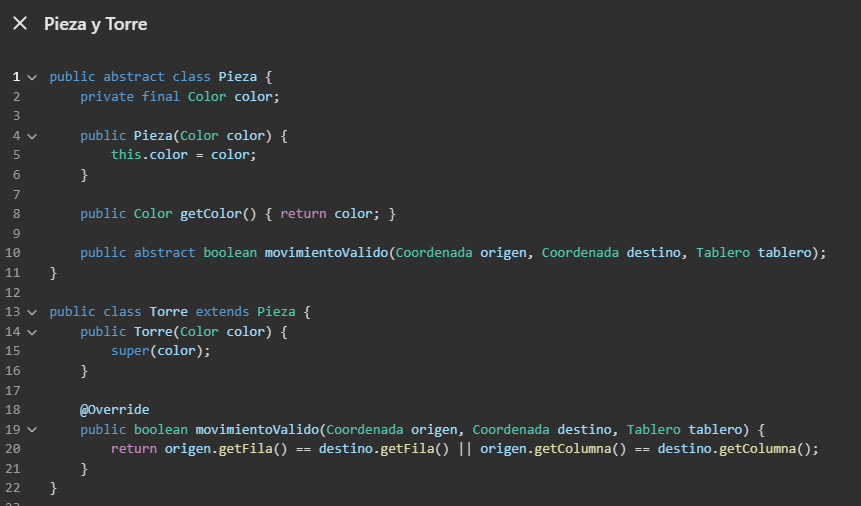


**🔍 ¿Por qué es un VO?**

* No tiene identidad propia.
* Se considera igual si la columna y fila son iguales.

# Herencia vs Composición

### 🎭 Opción 1: Usar **herencia** (clásico)



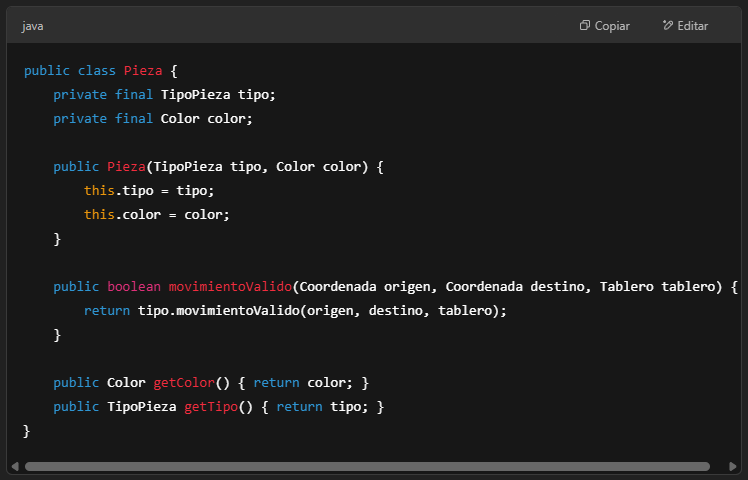
**✅ Ventajas:**

* Cada tipo de pieza **encapsula su propia lógica de movimiento**.
* Facilita el uso del polimorfismo (pieza.movimientoValido(...)).
* Código muy **natural desde la orientación a objetos**.

**⚠️ Desventajas:**

* La lógica puede dispersarse entre muchas clases.
* Agregar reglas específicas (como enroque, promoción, jaque mate) puede volverse engorroso si cada pieza lo maneja por su cuenta.
* Aumenta el acoplamiento si todas dependen de una misma estructura de Tablero.

### 🎲 Opción 2: Usar composición y tipo explícito



Y TipoPieza sería una especie de **estrategia**:



**✅ Ventajas:**

* Todo el comportamiento de las piezas está **centralizado en el enum** o en una clase de lógica.
* Fácil de testear y mantener.
* Permite trabajar con switch(tipo) en casos complejos (como IA o validaciones especiales).

**⚠️ Desventajas:**

* Pierdes un poco del sabor OOP.
* Requiere más diseño si quieres lógica muy específica por pieza.

# Enfoque con composición

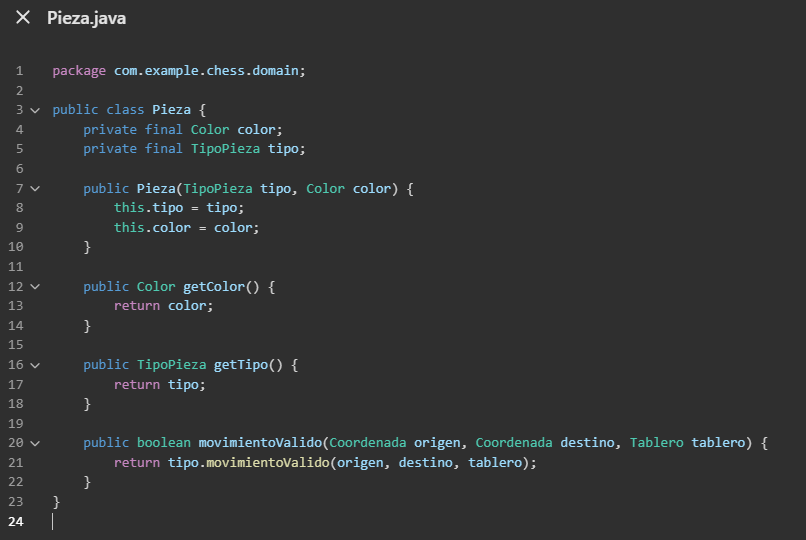
**🧩 Ventajas de esta composición:**

✅ Centraliza la lógica de piezas sin clases hijas  
✅ Facilita testeo, mantenimiento y extensión  
✅ Más flexible si necesitas agregar validaciones externas (jaque, obstrucciones, etc.)  
✅ Compatible con DDD porque el comportamiento **sigue embebido en el modelo del dominio**

## ♟️ Paso 1: Pieza

La idea es que:

* Una clase Pieza tenga un atributo TipoPieza, que encapsula la lógica de movimiento.
* En lugar de tener subclases como Torre, Caballo, etc., usamos una **enumeración inteligente** (TipoPieza) o una **estrategia** para definir el comportamiento de cada tipo.



## ♟️ Paso 2: TipoPieza con comportamiento embebido

**🧠 ¿Qué es TipoPieza?**

Un **enum con comportamiento**. Cada tipo de pieza implementa su propia lógica de movimiento.



# 🧱 Objetivo de Tablero

El Tablero será responsable de:

* Mantener la posición actual de todas las piezas (Map<Coordenada, Pieza>)
* Inicializar el estado inicial del juego
* Permitir mover piezas
* Consultar qué hay en una casilla

## 📦 Estructura de la clase Tablero



## 🧠 ¿Qué hace esta clase?

* inicial(): devuelve un tablero con unas piezas colocadas para comenzar (simplificado, pero puedes extenderlo).
* mover(): mueve la pieza si cumple su lógica y hay una pieza en la casilla de origen.
* obtenerPieza(): para consultar qué hay en una casilla.
* hayPiezaEn(): útil para saber si hay obstáculos.

# 🚀 ¿Qué sigue?

Ya tenemos:

✅ PartidaDeAjedrez (con la idea de orquestar todo)  
✅ Color, Jugador, Coordenada  
✅ Pieza y TipoPieza con composición  
✅ Tablero con estado, reglas básicas y movimientos