

@ChaloCode

DOMAIN DRIVEN DESIGN & CHATGPT



@ChaloCode

DDD estratégico



VS

DDD técnico

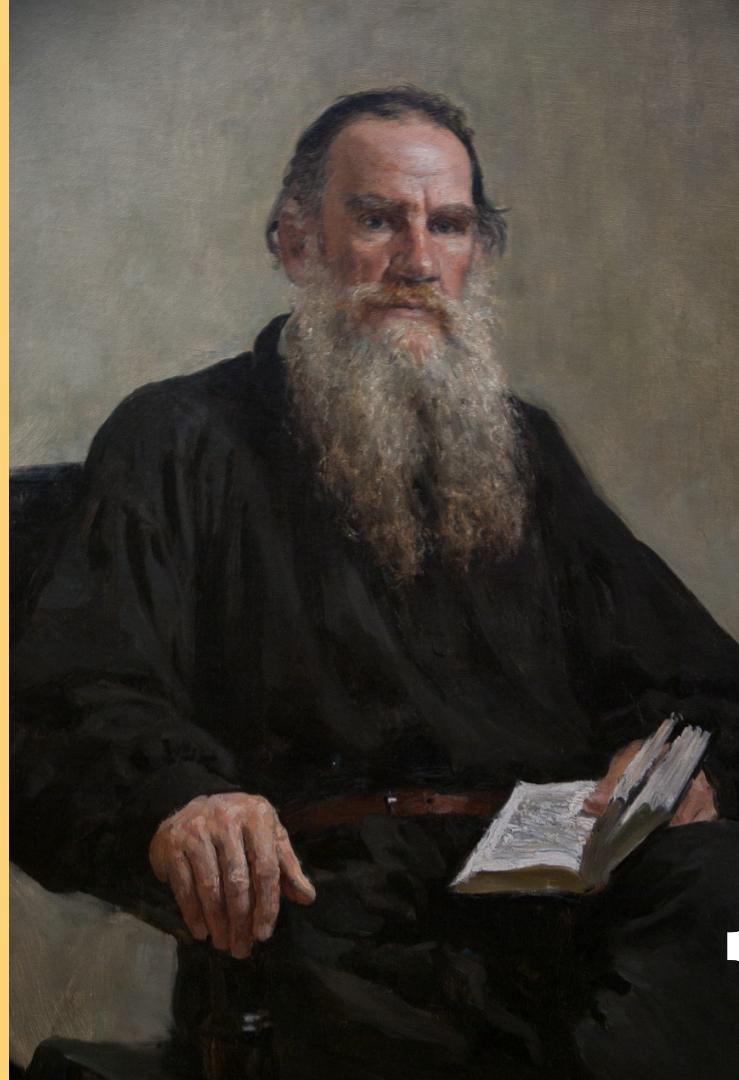


{C²}

Domain-Driven Design (DDD) es una metodología y un conjunto de principios y prácticas para diseñar y desarrollar software de manera que esté estrechamente alineado con el dominio o área problemática específica que el software está destinado a abordar.

@ChaloCode

¿ Si los
programadores
fuese
escritores
de novela ?



{C²}

@ChaloCode

DESVENTAJAS

*No todo
lo que brilla
es oro*



{C²}

@ChaloCode

Necesidad de expertos en el dominio

Complejidad inicial

Curva de aprendizaje

Sobreingeniería

Riesgo de separación de responsabilidades

Sobrediseño

Dificultad en proyectos pequeños

Mayor costo

Documentación excesiva

Resistencia cultural

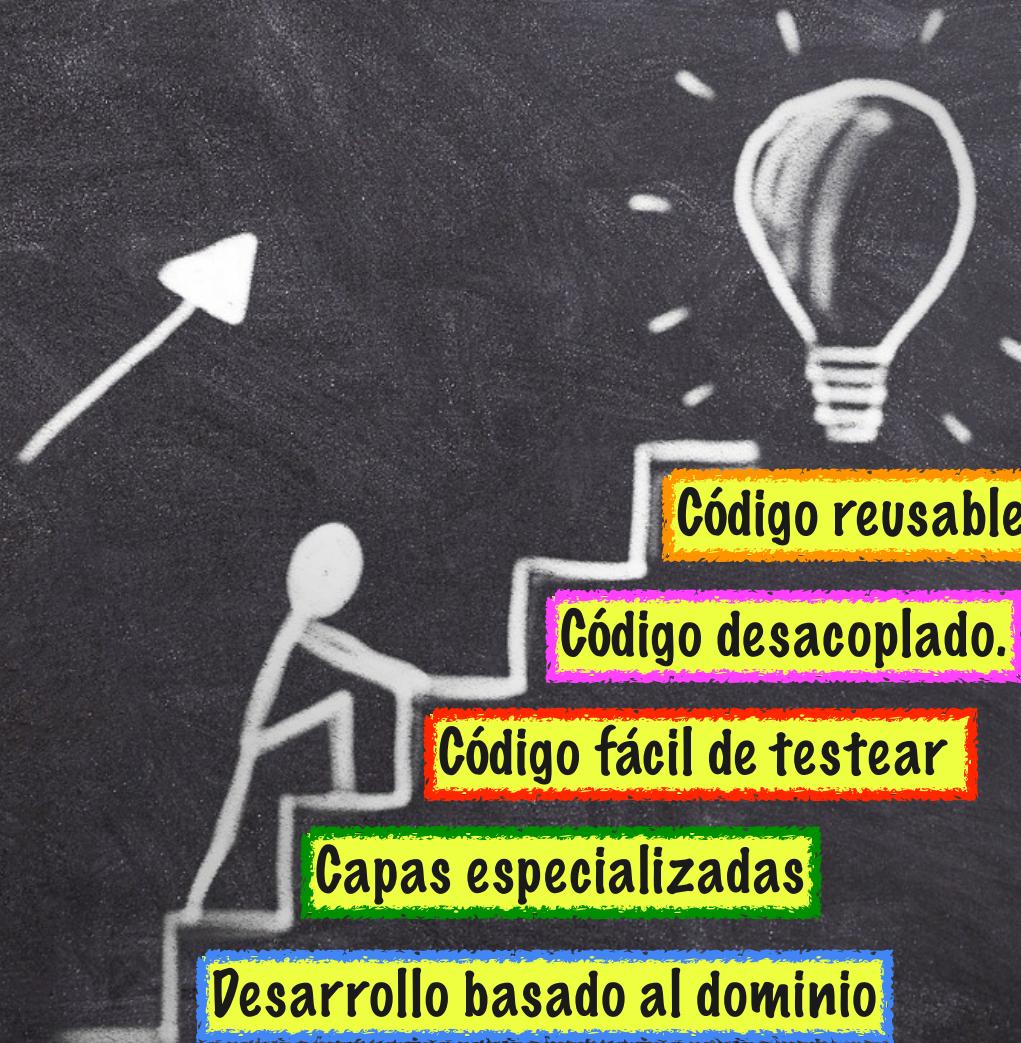
{C²}

BENEFICIOS

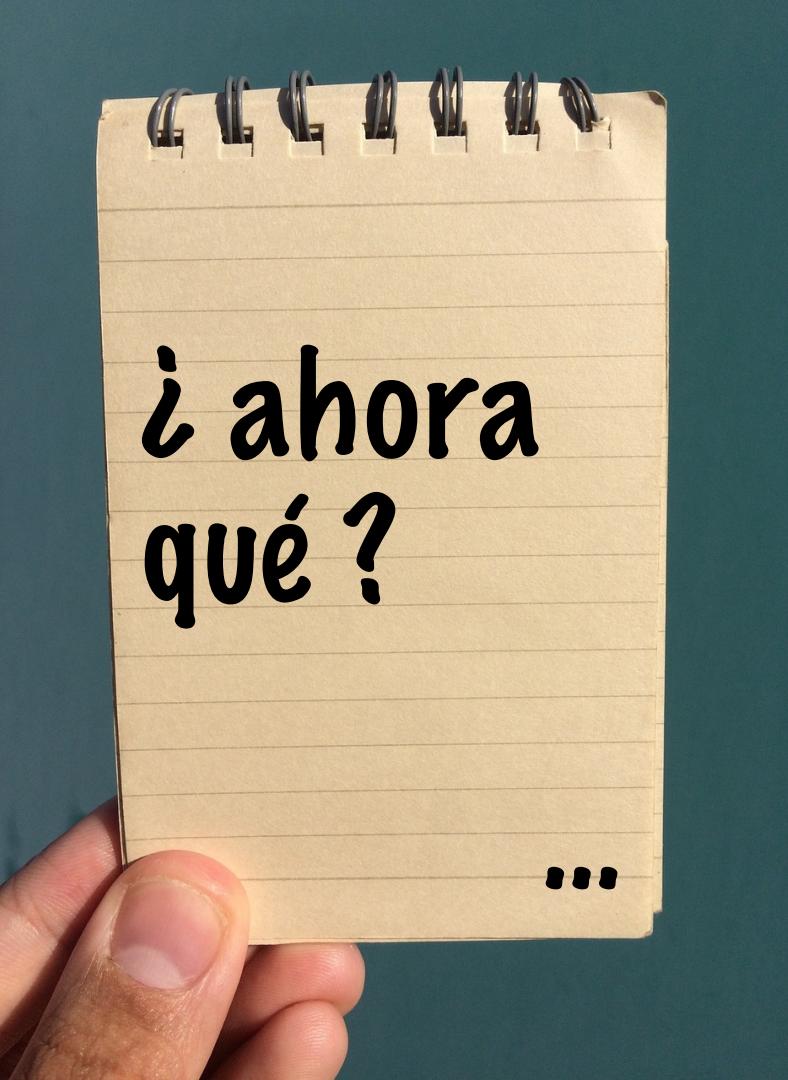


*El conocimiento
Es poder*

@ChaloCode



@ChaloCode



¿ahora
qué ?

{C²}

PONGÁMONOS SERIOS

HABLEMOS SOBRE

CAPAS DDD



Capa de Presentación (Presentation Layer):

Esta capa se encarga de interactuar con los usuarios y mostrar la información al exterior. Puede incluir interfaces de usuario, servicios web, controladores en una arquitectura MVC (Modelo-Vista-Controlador), y cualquier otra lógica relacionada con la presentación de la información.

Capa de Aplicación (Application Layer):

Aquí se encuentra la lógica de aplicación específica que coordina las interacciones entre la capa de presentación y la capa de dominio. Los servicios de aplicación y los casos de uso son típicos en esta capa. La capa de aplicación se encarga de traducir las acciones del usuario en operaciones en el dominio, orquestando las interacciones y aplicando reglas de negocio.

Capa de Dominio (Domain Layer):

Esta es la capa más importante en DDD, ya que contiene el modelo de dominio. Aquí es donde se define la lógica del negocio, los conceptos, las entidades, los agregados y las reglas específicas del dominio. La capa de dominio encapsula el conocimiento fundamental del problema que el software está abordando.

Capa de Infraestructura (Infrastructure Layer):

Esta capa proporciona las implementaciones concretas de los detalles técnicos que no pertenecen al dominio ni a la lógica de la aplicación. Puede incluir bases de datos, servicios de persistencia, servicios externos, frameworks, y cualquier otro componente técnico necesario para el funcionamiento del sistema. También puede haber adaptadores que conecten los componentes de infraestructura con el dominio.



¿abrumado?

Falta poco...

Siguiente: patrones tácticos

Entity:

Representa un objeto que tiene una identidad única a lo largo del tiempo y se distingue por sus atributos, no por sus valores. Por ejemplo, un "Cliente" podría ser una entidad.

Entity:



Podríamos considerar que cada tipo de cerveza es una entidad. Cada tipo de cerveza tiene características únicas, como el nombre, el estilo, el porcentaje de alcohol, etc. Cada entidad de tipo de cerveza tendría una identidad única a lo largo del tiempo.

Value Object:

Representa un objeto que no tiene identidad propia y se valora únicamente por sus atributos. Estos objetos son inmutables y se utilizan para representar conceptos como fechas, direcciones, coordenadas, etc.

Value Object:



Queremos representar las características específicas de una botella de cerveza, como su tamaño y color de tapa. Estos atributos no cambian el hecho de que es una botella de cerveza en particular, así que podríamos modelar estas características como objetos de valor inmutables.

Aggregate:

Un Aggregate es un grupo de entidades y value objects relacionados que se tratan como una única unidad coherente en las transacciones y reglas de negocio. Cada Aggregate tiene una raíz de agregado (Aggregate Root) que actúa como punto de acceso a las demás entidades y objetos del agregado.

Aggregate:



Podríamos definir un agregado para representar un tipo de cerveza y su inventario. Este agregado incluiría la entidad de tipo de cerveza como raíz del agregado y podría contener información sobre la cantidad de cerveza disponible en el inventario y las transacciones relacionadas con ese tipo de cerveza.

@ChaloCode

Repository:

El patrón Repository se utiliza para encapsular el acceso y la persistencia de las entidades. Proporciona una interfaz para acceder a las entidades sin exponer los detalles de almacenamiento subyacentes.

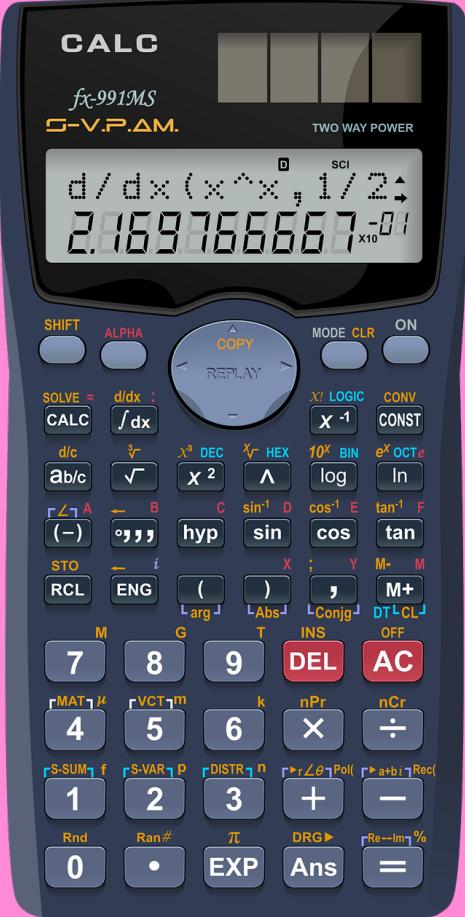
Repository:



Tendríamos un repositorio de tipos de cerveza que permitiría agregar, actualizar, eliminar y recuperar información sobre diferentes tipos de cerveza.

Service:

Los Services son componentes que realizan tareas o cálculos específicos que no encajan naturalmente en el modelo de un Entity o un Value Object. Ayudan a mantener la coherencia en el modelo al evitar que la lógica compleja se extienda por todo el código.



Service:

Supongamos que queremos calcular el nivel de existencias promedio de un tipo de cerveza en función de sus transacciones de entrada y salida. Podríamos crear un servicio de cálculo de existencias que tome en cuenta las transacciones y devuelva el nivel promedio de existencias.

@ChaloCode

Como hacerlo sin morir en el intento ?

Llama a un amigo o
usa ChatGPT



Me estafaron !!!

...y el código ?