12 Domain objects y casos de uso

¿Qué son los Domain Objects?

Los **Domain Objects** son los objetos que representan los conceptos principales y las reglas del **dominio del negocio** dentro de una aplicación. Son fundamentales en el enfoque de **Domain Driven Design (DDD)**, ya que encapsulan tanto los **datos** como los **comportamientos** relevantes para el dominio. Estos objetos son clave para asegurar que el sistema refleje correctamente las realidades y las reglas del negocio que modela.

1.1 Definición de Domain Objects

Un **Domain Object** es cualquier objeto que forma parte del modelo de dominio y que tiene un significado importante dentro del contexto del negocio. Estos objetos no solo contienen atributos o datos, sino también las **reglas de negocio** que determinan su comportamiento. Por ejemplo, en un sistema de comercio electrónico, objetos como `Pedido`, `Cliente` y `Producto` serían ejemplos típicos de Domain Objects.

- **Datos y comportamiento:** Los Domain Objects no son simples estructuras de datos; también definen comportamientos específicos. Por ejemplo, un `Pedido` no solo almacena información sobre los productos, sino que también define reglas para cómo puede ser completado o cancelado.
- **Reglas del dominio:** Las reglas que definen cómo un objeto puede cambiar de estado o interactuar con otros objetos son parte integral de los Domain Objects. Esto incluye restricciones, validaciones y políticas del negocio.

1.2 Tipos de Domain Objects: Entidades y Objetos de Valor

Dentro del contexto de DDD, los **Domain Objects** se dividen principalmente en dos tipos: **Entidades** y **Objetos de Valor (Value Objects)**.

- **Entidades:**
    - Las **Entidades** son Domain Objects que tienen una **identidad única** que persiste a lo largo del tiempo. Esta identidad es lo que define una entidad, más allá de los cambios en sus atributos.
    - **Ejemplo:** Un `Pedido` es una entidad porque tiene un identificador único (por ejemplo, un número de pedido) que lo distingue de otros pedidos. Aunque el contenido o el estado del pedido pueda cambiar, su identidad sigue siendo la misma.
    - En un proyecto Spring, las entidades suelen ser clases anotadas con `@Entity` y mapeadas a una base de datos usando **JPA (Java Persistence API)**.
- **Objetos de Valor (Value Objects):**

    - Los **Objetos de Valor** son Domain Objects que se definen únicamente por sus **atributos**. A diferencia de las entidades, no tienen identidad propia. Dos objetos de valor con los mismos atributos se consideran iguales.
    - **Ejemplo:** Una `Dirección` podría ser un Objeto de Valor en un sistema de pedidos. Si dos direcciones tienen los mismos valores para los atributos (calle, ciudad, código postal), se consideran la misma dirección.
    - Los Objetos de Valor suelen ser **inmutables**, lo que significa que, una vez creados, sus valores no cambian. Si es necesario cambiar un valor, se crea un nuevo objeto.

1.3 Ejemplos de Domain Objects en diferentes dominios

Dependiendo del dominio del negocio que se esté modelando, los Domain Objects variarán. Aquí algunos ejemplos comunes de Domain Objects en diferentes contextos:

- **Ecommerce:**
    - **Entidad:** `Pedido` (tiene un identificador único, se puede actualizar su estado).
    - **Objeto de Valor:** `Dirección` (se define por sus atributos y es inmutable).
- **Banca:**
    - **Entidad:** `CuentaBancaria` (tiene un número de cuenta único y el saldo puede cambiar).
    - **Objeto de Valor:** `Monto` (una cantidad de dinero, que es inmutable y se define solo por su valor numérico).
- **Logística:**
    - **Entidad:** `Envio` (cada envío tiene un identificador único y un estado de seguimiento que cambia con el tiempo).
    - **Objeto de Valor:** `CoordenadasGPS` (representa una ubicación geográfica que no cambia).

1.4 Relación con el modelo de dominio

Los **Domain Objects** forman el núcleo del **modelo de dominio**. En DDD, el modelo de dominio es una abstracción de la realidad del negocio, y los Domain Objects son las representaciones concretas de los elementos clave dentro de esa realidad. Cada Domain Object está relacionado de alguna manera con otros objetos dentro del dominio, y estas relaciones ayudan a definir cómo se estructura el sistema.

- **Interacción entre Domain Objects:** Los Domain Objects no actúan de forma aislada. Interactúan entre sí para cumplir las necesidades del negocio. Por ejemplo, un `Pedido` puede estar relacionado con múltiples `Producto`s y un `Cliente`. Estos objetos interactúan de acuerdo con las reglas definidas en el dominio.
- **Patrones tácticos de DDD:** En un modelo de dominio bien diseñado, las **entidades** y los **objetos de valor** son gestionados mediante **agregados** y **servicios de dominio**, lo que garantiza que el comportamiento y los estados de los objetos sean consistentes y correctos.

1.5 Dominio independiente de la infraestructura

Uno de los principios clave en la arquitectura hexagonal y en DDD es que los **Domain Objects** deben ser **independientes** de la infraestructura tecnológica (bases de datos, APIs, sistemas de mensajería). La lógica de negocio y los Domain Objects no deben verse afectados por los detalles técnicos del sistema.

- **Independencia tecnológica:** Los Domain Objects no deberían depender directamente de los detalles de persistencia (como bases de datos) o de la forma en que se comunican con sistemas externos. Esto se logra mediante **puertos** y **adaptadores**, que permiten que el dominio se mantenga autónomo y los detalles técnicos se gestionen de manera externa.

¿Para qué sirven los Domain Objects?

Los **Domain Objects** son esenciales para modelar la lógica de negocio de una aplicación. En el contexto de **Domain Driven Design (DDD)**, los Domain Objects ayudan a representar de manera precisa y detallada los conceptos del negocio y sus interacciones. A través de ellos, se pueden encapsular tanto los datos como las reglas y comportamientos del sistema, lo que asegura que las operaciones del software respeten las reglas del dominio.

2.1 Representación del dominio del negocio

El propósito principal de los **Domain Objects** es modelar los elementos clave del negocio en términos de datos y comportamientos. Estos objetos permiten que el código de la aplicación sea una representación directa de la realidad del negocio.

- **Simulación del mundo real:** Cada Domain Object en el sistema tiene su equivalente en el negocio. Por ejemplo, un `Pedido` en el negocio es modelado como una entidad `Pedido` en el sistema. Esto garantiza que el código sea coherente con las reglas y procesos del negocio.
- **Alineación con los expertos en el dominio:** Al usar Domain Objects que reflejan fielmente los términos y conceptos utilizados por los expertos del dominio (como clientes, productos, transacciones), se facilita la colaboración entre los desarrolladores y el negocio, ya que ambos trabajan con el mismo modelo conceptual.

2.2 Encapsulación de datos y comportamiento

Los **Domain Objects** no solo contienen datos, sino también las reglas y comportamientos relacionados con esos datos. Esto significa que encapsulan la lógica de negocio y las validaciones necesarias para que las entidades y objetos de valor operen correctamente.

- **Comportamientos alineados con el dominio:** Un **Domain Object** no solo almacena información, sino que también define cómo se pueden modificar los datos de acuerdo con las reglas del negocio. Por ejemplo, un `Pedido` puede tener un método `completarPedido()`, que verifica si se cumplen las condiciones necesarias para que el pedido sea completado (como la existencia de productos y un estado válido).

- **Validaciones internas:** Los Domain Objects se aseguran de que siempre se mantenga la **integridad** y la **consistencia** de los datos. Cualquier cambio en sus atributos debe pasar por las validaciones correspondientes antes de ser aceptado, lo que evita que se rompan las reglas del negocio.

2.3 Mantener la coherencia en las reglas del negocio

Una de las funciones más importantes de los **Domain Objects** es garantizar que las **reglas del negocio** se mantengan consistentes en todo el sistema. Estas reglas, también conocidas como **invariantes**, son las condiciones que deben cumplirse siempre para que el sistema funcione correctamente según las necesidades del negocio.

- **Reglas integradas en los objetos:** Las reglas de negocio se implementan directamente en los Domain Objects, asegurando que cualquier interacción con ellos respete dichas reglas. Esto significa que las operaciones realizadas sobre los objetos (como cambiar el estado de un pedido o calcular el total de una factura) se harán de manera controlada y conforme a las políticas del negocio.
- **Consistencia a través del sistema:** Los Domain Objects permiten que las reglas de negocio sean consistentes en todo el sistema, sin importar desde dónde se interactúe con los objetos (ya sea desde una API REST, un servicio interno, o un proceso de mensajería).

2.4 Abstracción de los detalles técnicos

Los **Domain Objects** permiten que la lógica de negocio esté desacoplada de los detalles técnicos del sistema. Esto significa que los Domain Objects no dependen de cómo los datos se persisten (por ejemplo, en una base de datos) o cómo se comunican con servicios externos. Este desacoplamiento es esencial para hacer que la lógica del negocio sea **independiente** y **reutilizable** en diferentes contextos.

- **Independencia de la infraestructura:** Gracias a la arquitectura hexagonal, los Domain Objects no conocen detalles sobre cómo interactúan con bases de datos o sistemas externos. Esto se maneja a través de **puertos** y **adaptadores**, lo que asegura que la lógica del dominio esté libre de dependencias tecnológicas.
- **Facilita los cambios tecnológicos:** Al mantener los Domain Objects libres de dependencias tecnológicas, es más fácil cambiar tecnologías o infraestructuras en el futuro sin afectar la lógica del negocio.

2.5 Reutilización del código y modularidad

Los **Domain Objects** son piezas fundamentales para promover la **reutilización del código** y mejorar la **modularidad** en el sistema. Al encapsular el comportamiento del dominio en objetos bien definidos, se facilita su reutilización en diferentes partes de la aplicación o incluso en proyectos futuros.

- **Reutilización en múltiples casos de uso:** Un mismo Domain Object puede ser utilizado en diferentes **casos de uso** o servicios dentro del sistema. Por ejemplo, un objeto `Cliente` puede ser utilizado tanto en el contexto de la gestión de pedidos como en el de la facturación, sin necesidad de duplicar lógica.

- **Modularidad:** Al estructurar el sistema en torno a Domain Objects que son independientes entre sí, se promueve la **modularidad**. Cada objeto encapsula su propia lógica, lo que facilita la separación de responsabilidades y el mantenimiento del código.

2.6 Facilitar las pruebas unitarias y la validación del negocio

Al encapsular las reglas de negocio dentro de los **Domain Objects**, se simplifica el proceso de pruebas unitarias. Estos objetos se pueden probar de manera aislada, ya que no dependen de infraestructura externa. Las pruebas sobre Domain Objects permiten validar fácilmente el cumplimiento de las reglas del negocio.

- **Pruebas unitarias sencillas:** Como los Domain Objects no tienen dependencias con bases de datos o sistemas externos, es posible realizar pruebas unitarias utilizando frameworks de pruebas como **JUnit**. Esto facilita la validación de comportamientos y reglas de negocio sin necesidad de configurar entornos complejos.
- **Validación del negocio:** Al tener las reglas encapsuladas dentro de los objetos, es más fácil **validar** que el sistema está funcionando conforme a los requisitos del negocio. Los cambios en los requisitos pueden implementarse directamente en los Domain Objects, y las pruebas aseguran que se cumplan las nuevas condiciones.

Creando nuestro primer Domain Object en un proyecto Spring bajo un modelo de arquitectura hexagonal

En un proyecto basado en la **arquitectura hexagonal** y **Spring**, los **Domain Objects** son esenciales para modelar la lógica del dominio. Siguiendo los principios de DDD y la estructura modular de la arquitectura hexagonal, los Domain Objects se ubican en el **núcleo del dominio**, manteniendo la independencia respecto a cualquier tecnología externa como bases de datos o APIs.

3.1 Paso 1: Definir la entidad o el objeto de valor

El primer paso en la creación de un **Domain Object** es determinar si estamos creando una **entidad** (con identidad única) o un **objeto de valor** (sin identidad, definido por sus atributos). En este ejemplo, crearemos una **entidad** llamada `Pedido`.

- **Entidad `Pedido`:** Es un Domain Object con una identidad única que representa una transacción de compra de productos dentro de un sistema de comercio electrónico. Cada `Pedido` tiene un identificador (`id`) y está compuesto por uno o varios productos que un cliente ha solicitado.

3.2 Paso 2: Agregar los atributos y comportamientos relevantes al Domain Object

En este paso, definimos los **atributos** que reflejan la información esencial del objeto y los **comportamientos** que encapsulan las reglas de negocio. En la arquitectura hexagonal, los Domain Objects no deben depender de detalles técnicos, por lo que su diseño se centra en representar las reglas del negocio.

- **Atributos de `Pedido`:**
  - `id`: Identificador único del pedido.
  - `cliente`: Información sobre el cliente que realizó el pedido.
  - `productos`: Lista de productos incluidos en el pedido.
  - `estado`: Estado actual del pedido (por ejemplo, PENDIENTE, COMPLETADO, CANCELADO).
- **Comportamientos del `Pedido`:**
  - `completarPedido()`: Método para cambiar el estado de un pedido a COMPLETADO, siempre que cumpla con ciertas condiciones (por ejemplo, que tenga productos asignados).
  - `cancelarPedido()`: Permite cancelar el pedido y cambiar su estado a CANCELADO.

Ejemplo de implementación del Domain Object `Pedido`:

```
public class Pedido {

    private Long id;
    private Cliente cliente;
    private List<Producto> productos;
    private EstadoPedido estado;

    public Pedido(Cliente cliente, List<Producto> productos) {
        this.cliente = cliente;
        this.productos = productos;
        this.estado = EstadoPedido.PENDIENTE;
    }

    // Comportamiento para completar el pedido
    public void completarPedido() {
        if (!productos.isEmpty()) {
            this.estado = EstadoPedido.COMPLETADO;
        } else {
            throw new IllegalStateException("No se puede completar un
pedido sin productos.");
        }
    }

    // Comportamiento para cancelar el pedido
    public void cancelarPedido() {
        if (this.estado == EstadoPedido.PENDIENTE) {
            this.estado = EstadoPedido.CANCELADO;
        } else {
            throw new IllegalStateException("Solo se puede cancelar un
```

```
pedido pendiente.");
        }
    }

    // Getters y setters
    public Long getId() {
        return id;
    }

    public Cliente getCliente() {
        return cliente;
    }

    public List<Producto> getProductos() {
        return productos;
    }

    public EstadoPedido getEstado() {
        return estado;
    }
}
```

3.3 Paso 3: Asegurar el desacoplamiento de la infraestructura

Uno de los principios clave de la arquitectura hexagonal es que los **Domain Objects** deben estar desacoplados de la infraestructura técnica. Esto significa que el `Pedido` no debe tener ningún conocimiento sobre cómo se almacena en una base de datos o cómo se interactúa con sistemas externos. Para lograr este desacoplamiento, los Domain Objects interactúan con el mundo exterior a través de **puertos** (interfaces) que se implementan mediante **adaptadores**.

- **Puertos para persistencia:** En lugar de que el Domain Object `Pedido` sepa cómo persistirse en una base de datos, define una interfaz (puerto) que describe cómo se guarda y recupera, como `RepositorioDePedidos`. El puerto será implementado por un adaptador externo, por ejemplo, utilizando **Spring Data** para la base de datos.

Ejemplo de puerto `RepositorioDePedidos`:

```
public interface RepositorioDePedidos {
    Pedido guardar(Pedido pedido);
    Optional<Pedido> buscarPorId(Long id);
}
```

En esta interfaz, se definen los métodos que permiten almacenar y recuperar pedidos sin que el Domain Object tenga conocimiento de cómo se implementan.

3.4 Integración con Spring Boot utilizando puertos y adaptadores

En un proyecto **Spring Boot**, los puertos y adaptadores se integran de forma eficiente, manteniendo la independencia del dominio. El **adaptador de salida** que implementa el puerto `RepositorioDePedidos` puede usar **Spring Data JPA** para interactuar con la base de datos, pero el Domain Object `Pedido` no necesita saber nada sobre JPA.

Ejemplo de implementación del adaptador de salida `RepositorioDePedidosImpl`:

```
@Repository
public class RepositorioDePedidosImpl implements RepositorioDePedidos {

    @Autowired
    private JpaPedidoRepository jpaPedidoRepository;

    @Override
    public Pedido guardar(Pedido pedido) {
        return jpaPedidoRepository.save(pedido);
    }

    @Override
    public Optional<Pedido> buscarPorId(Long id) {
        return jpaPedidoRepository.findById(id);
    }
}
```

En este ejemplo, el **adaptador** utiliza **JpaPedidoRepository** (que extiende de **JpaRepository**) para gestionar la persistencia de los pedidos, pero el Domain Object `Pedido` permanece completamente desacoplado de estos detalles.

3.5 Ventajas de utilizar Domain Objects en la arquitectura hexagonal

- **Independencia del dominio:** Los Domain Objects se enfocan únicamente en las reglas de negocio y no dependen de los detalles técnicos o de infraestructura.
- **Fácilmente testeables:** Al estar desacoplados de la infraestructura externa, los Domain Objects se pueden probar de manera aislada utilizando pruebas unitarias.
- **Modularidad:** El uso de puertos y adaptadores asegura que el sistema sea modular y fácilmente escalable, ya que los detalles de infraestructura pueden cambiar sin afectar la lógica del dominio.

¿Qué son los casos de uso en la arquitectura hexagonal?

En la **arquitectura hexagonal**, los **casos de uso** representan las operaciones de negocio que interactúan con los **Domain Objects** para realizar una tarea o proceso específico dentro del sistema. Un caso de uso encapsula una secuencia de acciones que responden a una necesidad del usuario o del sistema, y se implementa mediante **servicios de aplicación** o **servicios de dominio** que coordinan las interacciones entre los objetos del dominio y los puertos.

4.1 Definición de casos de uso

Los **casos de uso** en la arquitectura hexagonal son un punto de entrada a la lógica de negocio, donde se definen los pasos que el sistema debe seguir para ejecutar una tarea en respuesta a una solicitud. Estos casos de uso orquestan las interacciones entre los Domain Objects, los puertos y los adaptadores, gestionando la lógica necesaria para cumplir con las reglas del dominio.

- **Casos de uso centrados en el negocio:** Los casos de uso son operaciones específicas que representan una funcionalidad completa del negocio. No están preocupados por los detalles técnicos de la infraestructura, como la persistencia o las interfaces de usuario, sino que se enfocan en resolver un problema o cumplir un objetivo del negocio.
- **Ejemplo:** En un sistema de comercio electrónico, un caso de uso podría ser el proceso de **crear un pedido**, que involucraría la validación de los productos, la actualización del estado del pedido y la interacción con servicios externos para el procesamiento de pagos.

4.2 Casos de uso como servicios de aplicación

Los **servicios de aplicación** son la implementación de los casos de uso dentro de la arquitectura hexagonal. Estos servicios actúan como intermediarios entre los puertos de entrada (como controladores o interfaces de usuario) y el núcleo del dominio (donde se encuentran los Domain Objects y las reglas de negocio).

- **Intermediarios entre puertos y el dominio:** Los servicios de aplicación se encargan de recibir las solicitudes desde los puertos de entrada (por ejemplo, una API REST o una interfaz de usuario) y coordinar las operaciones sobre los Domain Objects a través de los puertos de salida (por ejemplo, repositorios, APIs de terceros).
- **Ejemplo de servicio de aplicación para gestionar pedidos:**
    - El servicio de aplicación `GestorDePedidos` recibe una solicitud para crear un pedido, valida los productos, calcula el total del pedido, guarda el pedido en un repositorio y envía una notificación de confirmación al cliente.

4.3 Función de los casos de uso en la arquitectura hexagonal

Los **casos de uso** tienen varias funciones importantes dentro de la arquitectura hexagonal:

- **Coordinar la lógica de negocio:** Los casos de uso orquestan las acciones que los **Domain Objects** deben realizar, siguiendo las reglas del negocio. Esto incluye validar datos de entrada, interactuar con diferentes objetos del dominio y asegurarse de que se mantengan las invariantes del sistema.
- **Desacoplamiento de la infraestructura técnica:** Al igual que los Domain Objects, los casos de uso no dependen de la infraestructura externa (como bases de datos o sistemas de mensajería). En cambio, interactúan con **puertos**, lo que les permite mantenerse independientes de los detalles técnicos y enfocarse solo en el negocio.
- **Encapsulamiento de la lógica de aplicación:** Los casos de uso encapsulan toda la lógica relacionada con una operación específica del negocio. Esto permite que la lógica sea más modular, fácil de entender y testeable. Cada caso de uso se puede probar de manera independiente para asegurarse de que cumple con las expectativas del negocio.

4.4 Ejemplos de casos de uso en sistemas reales

En un sistema real, los **casos de uso** representan operaciones completas que responden a las necesidades del usuario o del negocio. A continuación, se muestran algunos ejemplos de cómo los casos de uso se implementan en diferentes contextos.

- **Sistema de comercio electrónico:**
    - **Caso de uso:** Crear un pedido.
        - Recibe la información del cliente y los productos.
        - Valida que haya stock suficiente para cada producto.
        - Calcula el total del pedido y procesa el pago.
        - Guarda el pedido en un repositorio.
        - Actualiza el estado del pedido y envía una confirmación al cliente.
- **Sistema bancario:**
    - **Caso de uso:** Transferir fondos entre cuentas.
        - Valida que ambas cuentas existan y tengan saldo suficiente.
        - Actualiza los saldos de las cuentas involucradas.
        - Registra la transacción en un historial de movimientos.
        - Envía notificaciones a ambas partes involucradas.

4.5 Relación entre casos de uso y Domain Objects

Los **casos de uso** orquestan el comportamiento de los **Domain Objects** para ejecutar las reglas del negocio. Los Domain Objects son los encargados de manejar los datos y las operaciones básicas (por ejemplo, cambiar el estado de un pedido), mientras que los casos de uso se encargan de coordinar la interacción entre varios objetos del dominio y asegurarse de que la lógica del negocio se aplique correctamente.

- **Interacción entre casos de uso y Domain Objects:**
    - Un caso de uso como `crearPedido()` puede interactuar con varios Domain Objects, como `Cliente`, `Producto` y `Pedido`. Cada objeto maneja sus propias reglas y estados, mientras que el caso de uso se asegura de que todas las interacciones se realicen correctamente para completar la operación.

4.6 Uso de puertos para implementar casos de uso

En la arquitectura hexagonal, los **casos de uso** interactúan con el exterior a través de **puertos**. Los puertos definen las interfaces que permiten la entrada de solicitudes y la salida de resultados hacia el mundo exterior, desacoplando así la lógica de negocio de la infraestructura técnica.

- **Puertos de entrada:** Permiten que los actores externos (como usuarios o sistemas) inicien un caso de uso. Por ejemplo, un puerto de entrada puede ser una API REST que recibe una solicitud para crear un pedido.
- **Puertos de salida:** Los casos de uso utilizan puertos de salida para interactuar con servicios externos o infraestructuras técnicas (como bases de datos o servicios de mensajería). Un puerto de salida podría ser un repositorio que permite guardar o recuperar un pedido.

Ejemplo de puerto de entrada para un caso de uso en Spring:

```
public interface GestorDePedidos {
    Pedido crearPedido(Cliente cliente, List<Producto> productos);
    void cancelarPedido(Long pedidoId);
}
```

Este puerto define los métodos que el caso de uso `GestorDePedidos` expone para que otros sistemas interactúen con él, sin importar cómo se implementa la lógica interna.

¿Para qué sirven los casos de uso?

Los **casos de uso** en la **arquitectura hexagonal** tienen un papel fundamental en la organización y estructuración de las operaciones que el sistema debe realizar para cumplir con los requisitos del negocio. Estos casos de uso encapsulan la lógica de alto nivel, coordinan las interacciones entre los **Domain Objects** y aseguran que el sistema funcione de acuerdo con las reglas de negocio. Además, proporcionan una interfaz clara para interactuar con el dominio sin exponer detalles internos ni comprometer la flexibilidad del sistema.

5.1 Organizar y estructurar las acciones del negocio

Uno de los propósitos principales de los **casos de uso** es organizar y estructurar las **acciones del negocio** en operaciones concretas que el sistema pueda ejecutar. Los casos de uso definen los pasos y las reglas que se deben seguir para lograr un objetivo de negocio, asegurando que todo el proceso sea coherente y siga las políticas del dominio.

- **Organización de procesos complejos:** Los casos de uso simplifican la coordinación de procesos complejos que involucran múltiples interacciones con distintos **Domain Objects**. Por ejemplo, en el caso de un proceso de compra, el

caso de uso puede validar los productos, procesar el pago, actualizar el estado del pedido y notificar al cliente, todo en una única operación organizada.
- **Ejecución de reglas de negocio:** Cada caso de uso ejecuta las reglas de negocio que son relevantes para una operación particular. Las reglas son gestionadas dentro del caso de uso y aplicadas de manera coherente en cada interacción con el sistema.

5.2 Aislar la lógica de negocio de la infraestructura técnica

En la **arquitectura hexagonal**, los **casos de uso** actúan como un intermediario entre el núcleo del dominio y las infraestructuras externas, como bases de datos, APIs, interfaces de usuario o servicios de mensajería. Gracias a los **puertos** y **adaptadores**, los casos de uso pueden mantenerse completamente independientes de los detalles técnicos, lo que facilita su testabilidad y su evolución.

- **Desacoplamiento de la infraestructura:** Los casos de uso no interactúan directamente con la base de datos o los sistemas externos. En lugar de eso, utilizan puertos que actúan como interfaces, lo que permite que los casos de uso se enfoquen en la lógica del negocio y no en los detalles técnicos.
- **Facilidad de evolución:** Debido a que los casos de uso están desacoplados de la infraestructura técnica, es fácil cambiar las tecnologías subyacentes (por ejemplo, cambiar de una base de datos SQL a NoSQL) sin modificar la lógica del negocio encapsulada en los casos de uso.

5.3 Encapsular y proteger la lógica de negocio

Los casos de uso encapsulan la **lógica de negocio** de alto nivel y protegen la consistencia del sistema al garantizar que todas las interacciones sigan las reglas establecidas por el negocio. Al hacerlo, evitan que los detalles técnicos o infraestructurales interfieran con las reglas y políticas del dominio.

- **Protección del dominio:** Los casos de uso protegen el núcleo del dominio al asegurarse de que las reglas del negocio se apliquen correctamente antes de modificar los estados o atributos de los **Domain Objects**. Esto asegura que los **invariantes** (condiciones que siempre deben cumplirse) se mantengan, y que las operaciones que involucren múltiples objetos del dominio se realicen de manera segura y consistente.
- **Control de flujos de trabajo:** Al encapsular los flujos de trabajo del negocio, los casos de uso también controlan las secuencias de pasos que se deben seguir en una operación. Esto asegura que el sistema se comporte de manera predecible y que cualquier fallo sea detectado y gestionado de forma centralizada dentro del caso de uso.

5.4 Facilitar las pruebas unitarias y la validación de procesos

Otro beneficio importante de los **casos de uso** es que permiten realizar pruebas unitarias más simples y efectivas. Como los casos de uso están desacoplados de la infraestructura técnica y encapsulan la lógica de negocio, se pueden probar de manera aislada sin la necesidad de configurar entornos complejos como bases de datos o APIs externas.

- **Pruebas unitarias más sencillas:** Al estar desacoplados de la infraestructura, los casos de uso pueden ser probados utilizando mocks o stubs para simular las interacciones con puertos y adaptadores. Esto permite validar la lógica del negocio de manera eficiente y rápida.
- **Verificación del comportamiento del negocio:** Las pruebas de los casos de uso permiten validar que los procesos del negocio se ejecutan correctamente y que las reglas del negocio se aplican de manera consistente. Esto es fundamental para asegurar que el sistema se comporta conforme a los requisitos del cliente o del negocio.

5.5 Proporcionar una interfaz clara para el sistema

Los **casos de uso** también proporcionan una interfaz clara y definida para que los actores externos (ya sean usuarios, APIs, o sistemas externos) interactúen con el sistema. Al definir métodos bien estructurados que representan operaciones de negocio, los casos de uso ofrecen una forma sencilla y directa de realizar tareas específicas en el sistema sin exponer detalles innecesarios.

- **Interfaz de entrada bien definida:** Cada caso de uso ofrece un conjunto de métodos o acciones que representan tareas de negocio completas, lo que permite que los actores externos interactúen con el sistema de manera intuitiva.
- **Facilidad de mantenimiento:** Al proporcionar una interfaz clara y consistente, los casos de uso facilitan el mantenimiento y la evolución del sistema. Si las reglas del negocio cambian, solo es necesario actualizar el caso de uso correspondiente, manteniendo la interfaz intacta.

5.6 Modularidad y flexibilidad en el diseño

Los **casos de uso** permiten crear un diseño **modular** y **flexible**, donde cada operación de negocio está separada de otras, pero sigue las mismas reglas de interacción con los **Domain Objects** y la infraestructura técnica. Esta modularidad permite que el sistema crezca y evolucione sin que las modificaciones en un caso de uso afecten a otros.

- **Modularidad:** Cada caso de uso se puede desarrollar, probar y desplegar de manera independiente. Esta modularidad mejora la capacidad de escalar el sistema y permite que diferentes equipos trabajen en diferentes casos de uso sin interferir entre sí.
- **Flexibilidad:** Los casos de uso pueden evolucionar fácilmente en respuesta a cambios en los requisitos del negocio, sin afectar otros casos de uso o la estructura global del sistema. Esto permite una gran flexibilidad y adaptabilidad en sistemas que necesitan ajustarse a cambios frecuentes.

Creación de proyecto Spring bajo un modelo de arquitectura hexagonal

La **arquitectura hexagonal** se adapta perfectamente al desarrollo de aplicaciones con **Spring Boot** debido a su enfoque modular y su capacidad para desacoplar la lógica de negocio de los detalles técnicos. Crear un proyecto con esta arquitectura implica organizar

el código en módulos claros, donde el dominio (las reglas del negocio) esté completamente separado de las infraestructuras externas (bases de datos, APIs, servicios de terceros). Esto se logra a través de puertos y adaptadores, lo que asegura que el sistema sea fácil de probar, mantener y escalar.

6.1 Paso 1: Definir los Domain Objects en el núcleo del dominio

El primer paso en la creación de un proyecto hexagonal es definir los **Domain Objects** en el núcleo del dominio. Estos objetos representan las entidades clave del sistema y encapsulan tanto los datos como los comportamientos que siguen las reglas del negocio.

- **Crear entidades y objetos de valor:** Los **Domain Objects** incluyen tanto **entidades** (con identidad única) como **objetos de valor** (definidos por sus atributos). Estos objetos son el núcleo de la aplicación y deben estar completamente desacoplados de los detalles técnicos.

Ejemplo de una entidad `Pedido`:

```
public class Pedido {
    private Long id;
    private Cliente cliente;
    private List<Producto> productos;
    private EstadoPedido estado;

    public Pedido(Cliente cliente, List<Producto> productos) {
        this.cliente = cliente;
        this.productos = productos;
        this.estado = EstadoPedido.PENDIENTE;
    }

    public void completarPedido() {
        if (!productos.isEmpty()) {
            this.estado = EstadoPedido.COMPLETADO;
        } else {
            throw new IllegalStateException("No se puede completar un
pedido sin productos.");
        }
    }

    // Getters y setters
}
```

6.2 Paso 2: Crear los puertos que actúan como interfaces para interactuar con el sistema

En la **arquitectura hexagonal**, los **puertos** son interfaces que definen cómo el núcleo del sistema interactúa con el exterior, ya sea para recibir solicitudes (puertos de entrada) o para interactuar con infraestructuras técnicas (puertos de salida).

- **Puertos de entrada:** Definen las operaciones que los actores externos pueden realizar en el sistema. En el contexto de Spring Boot, estos puertos pueden ser utilizados por controladores REST u otros adaptadores de entrada.
- **Puertos de salida:** Definen las operaciones que el núcleo del dominio requiere para interactuar con sistemas externos, como repositorios de bases de datos o servicios de mensajería.

Ejemplo de puerto de entrada `GestorDePedidos`:

```
public interface GestorDePedidos {
    Pedido crearPedido(Cliente cliente, List<Producto> productos);
    void cancelarPedido(Long pedidoId);
}
```

Ejemplo de puerto de salida `RepositorioDePedidos`:

```
public interface RepositorioDePedidos {
    Pedido guardar(Pedido pedido);
    Optional<Pedido> buscarPorId(Long id);
}
```

6.3 Paso 3: Implementar los casos de uso para manejar la lógica del negocio

Los **casos de uso** encapsulan las operaciones que el sistema debe realizar para cumplir con los objetivos del negocio. En el contexto de Spring Boot, los casos de uso son implementados como **servicios de aplicación** que coordinan las interacciones entre los Domain Objects y los puertos.

- **Casos de uso como servicios de aplicación:** Estos servicios reciben las solicitudes desde los adaptadores de entrada (por ejemplo, controladores REST), ejecutan las reglas del negocio sobre los **Domain Objects**, y utilizan los puertos de salida para interactuar con las infraestructuras técnicas.

Ejemplo de implementación del caso de uso `GestorDePedidosImpl`:

```
@Service
public class GestorDePedidosImpl implements GestorDePedidos {
```

```
    private final RepositorioDePedidos repositorioDePedidos;

    @Autowired
    public GestorDePedidosImpl(RepositorioDePedidos
repositorioDePedidos) {
        this.repositorioDePedidos = repositorioDePedidos;
    }

    @Override
    public Pedido crearPedido(Cliente cliente, List<Producto> productos)
{
        Pedido pedido = new Pedido(cliente, productos);
        return repositorioDePedidos.guardar(pedido);
    }

    @Override
    public void cancelarPedido(Long pedidoId) {
        Pedido pedido = repositorioDePedidos.buscarPorId(pedidoId)
                .orElseThrow(() -> new IllegalArgumentException("Pedido
no encontrado"));
        pedido.cancelarPedido();
        repositorioDePedidos.guardar(pedido);
    }
}
```

6.4 Paso 4: Desarrollar los adaptadores de entrada y salida

Los **adaptadores** son las implementaciones concretas de los puertos que permiten que el sistema interactúe con el mundo externo, ya sea mediante **controladores REST** (adaptadores de entrada) o mediante la **persistencia de datos** (adaptadores de salida).

- **Adaptadores de entrada:** Estos adaptadores implementan los puertos de entrada y permiten que los actores externos (como usuarios o sistemas) interactúen con el sistema. En un proyecto Spring Boot, estos adaptadores suelen ser **controladores REST**.
- **Adaptadores de salida:** Estos adaptadores implementan los puertos de salida y permiten que el núcleo del sistema interactúe con servicios externos, como bases de datos o APIs de terceros.

Ejemplo de adaptador de entrada `PedidoController`:

```
@RestController
@RequestMapping("/pedidos")
public class PedidoController {
```

```
    private final GestorDePedidos gestorDePedidos;

    @Autowired
    public PedidoController(GestorDePedidos gestorDePedidos) {
        this.gestorDePedidos = gestorDePedidos;
    }

    @PostMapping
    public Pedido crearPedido(@RequestBody PedidoRequest request) {
        return gestorDePedidos.crearPedido(request.getCliente(),
request.getProductos());
    }
}
```

Ejemplo de adaptador de salida `RepositorioDePedidosImpl` utilizando Spring Data:

```
@Repository
public class RepositorioDePedidosImpl implements RepositorioDePedidos {

    @Autowired
    private JpaPedidoRepository jpaPedidoRepository;

    @Override
    public Pedido guardar(Pedido pedido) {
        return jpaPedidoRepository.save(pedido);
    }

    @Override
    public Optional<Pedido> buscarPorId(Long id) {
        return jpaPedidoRepository.findById(id);
    }
}
```

6.5 Paso 5: Utilizar Spring Data para gestionar la persistencia

En la **arquitectura hexagonal**, la persistencia de los **Domain Objects** se maneja a través de los puertos de salida, que son implementados por adaptadores de persistencia. **Spring Data JPA** facilita esta tarea proporcionando una capa de abstracción para interactuar con bases de datos de manera sencilla.

- **Repositorio JPA:** Spring Data JPA permite definir repositorios que gestionan la persistencia de las entidades sin necesidad de escribir grandes cantidades de código de acceso a datos.

Ejemplo de repositorio JPA `JpaPedidoRepository`:

```
public interface JpaPedidoRepository extends JpaRepository<Pedido, Long>
{
}
```

6.6 Ventajas de la arquitectura hexagonal en un proyecto Spring Boot

- **Desacoplamiento de la lógica de negocio:** La lógica del dominio permanece desacoplada de los detalles técnicos, lo que facilita el mantenimiento y la evolución del sistema.
- **Testabilidad:** Al estar los casos de uso y los Domain Objects desacoplados de la infraestructura, se pueden probar de manera aislada mediante pruebas unitarias.
- **Modularidad y flexibilidad:** El uso de puertos y adaptadores permite cambiar o reemplazar tecnologías sin afectar la lógica del negocio. Por ejemplo, cambiar la base de datos de SQL a NoSQL no afectaría al núcleo del dominio.
- **Escalabilidad:** La arquitectura hexagonal facilita la creación de sistemas modulares y escalables, ideales para arquitecturas basadas en microservicios.